ASP.NET Core 2

Guida completa per lo sviluppatore

Daniele Bochicchio, Cristian Civera
Moreno Gentili, Stefano Mostarda, Matteo Tumiati

aspitalia.com
The ASP.NET Community

HOEPLI
INFORMATICA

Daniele Bochicchio Cristian Civera Moreno Gentili Stefano Mostarda Matteo Tumiati

ASP.NET Core 2

Guida completa per lo sviluppatore

EDITORE ULRICO HOEPLI MILANO

Indice

Contenuti del libro

ASPItalia.com Network

Gli autori

Capitolo 1 - Primi passi con .NET Core e ASP.NET Core

Genesi di .NET Core

Analisi di .NET Core

Creare applicazioni per .NET Core

ASP.NET Core

ASP.NET Core e .NET Framework

.NET Standard

Scegliere .NET Core o .NET Framework

Conclusioni

Capitolo 2 - Usare .NET Core CLI, Visual Studio e Visual Studio Code per iniziare con ASP.NET

Scegliere l'ambiente di sviluppo

IDE o editor?

Installare il .NET Core SDK

Gestire progetti ASP.NET Core con il .NET Core CLI

Guida all'utilizzo del comando dotnet

Selezionare una versione del .NET Core SDK

Elencare le SDK installate e determinare quella in uso

Cambiare la versione del .NET Core SDK con il global.json

Il primo progetto con ASP.NET Core

Scegliere un template di progetto

Creare un progetto dagli ambienti di sviluppo

Installare altri template di progetto

Compilare ed eseguire l'applicazione

Le applicazioni ASP.NET Core sono applicazioni console

Debug dell'applicazione

Debug con Visual Studio

Debug con Visual Studio Code

Debug con Visual Studio for Mac

Creare una solution con molteplici progetti

Creare la solution

Aggiungere un progetto di tipo Class Library

Aggiungere un progetto di unit testing

Lavorare con i pacchetti NuGet

Aggiungere pacchetti NuGet al progetto

Creare un nostro personale pacchetto NuGet

Preparare l'applicazione per la distribuzione

Il comando dotnet publish

Cross-compilazione e pubblicazione self-contained

Creare uno store a livello di sistema dei pacchetti NuGet
Avviare l'applicazione nella macchina server

Conclusioni

Capitolo 3 - Anatomia di un progetto ASP.NET Core

Una visione d'insieme del progetto ASP.NET Core

Le directory del progetto

La directory wwwroot
Organizzare il codice in directory
Le directory obj e bin

La classe Program

Personalizzare il web host
Indicare gli endpoint
Scegliere un web server: Kestrel o HTTP.sys?
Cambiare il percorso delle directory principali
Impostare un environment
Configurare il web host da fonti esterne
Scegliere la classe di configurazione dell'applicazione

Configurare l'applicazione con la classe Startup

Usare middleware dal metodo Configure

Come funzionano i middleware

Aggiungere servizi dal metodo ConfigureServices

Cos'è un servizio

Configurare l'applicazione

Variare la configurazione in base all'environment
Leggere la configurazione in maniera fortemente tipizzata

Che fine hanno fatto il web.config e il global.asax?

Il file .csproj

Il metapacchetto Microsoft.AspNetCore.App

Il tool Microsoft.VisualStudio.Web.CodeGeneration.Tools

Conclusioni

Capitolo 4 - Primi passi con ASP.NET Core MVC

Il pattern Model-View-Controller

I servizi di routing

Il controller e il model

La view e gli HTML helper

Gestire una form di input

ASP.NET Core MVC e progetti complessi: le aree

Conclusioni

Capitolo 5 - Controller e routing

URL Routing in ASP.NET Core MVC

Constraint sulle route

Anatomia di un controller

Proprietà e metodi di supporto della classe Controller

Ciclo di vita di un controller

Uso della Dependency Injection nei controller

Utilizzare un motore di IoC container differente

La action come gestore della richiesta

L'interfaccia IActionResult e i diversi tipi di risposta

I tipi ViewResult e PartialViewResult
I tipi RedirectResult, RedirectToRouteResult e HttpStatusCodeResult
Restituire JSON con JsonResult
Il tipo FileResult e il metodo File
Il tipo ContentResult e il metodo Content
Controllo dell'esecuzione di una action
Esecuzione asincrona di una action
L'infrastruttura dei filtri
Conclusioni
Capitolo 6 - Le view e Razor
Creare view in ASP.NET Core MVC grazie a Razor
La sintassi di base
Branch e cicli
Definire funzioni in una view
Le view e il model: tipizzazione debole e forte
Consistenza grafica tra le pagine: la layout view
Sfruttare le layout view nel progetto
La view _ViewStart
Definire sezioni aggiuntive in una layout view
Semplificare il codice delle view: gli HTML e i tag helper
Gli HTML Helper
Il metodo ActionLink
Il metodo RouteLink
Il metodo Raw
Il metodo Partial e le partial view
I Tag helper e la gestione del markup

Il Tag helper Partial
Il Tag helper Environment
Il Tag helper Anchor
Il Tag helper Image
Conclusioni

Capitolo 7 - Gestione delle form

Generare la form da un model
Definire il model
L'attributo Display
L'attributo DisplayFormat
L'attributo UIHint
Definire il controller e la action
Creare la view
Il tag helper Form
Il tag helper Input
Corrispondenza tra tipi e tag Helper
Il tag helper Select
Il tag helper Textarea
Il tag helper Label
Gli HTML helper Editor e EditorFor
Gli HTML helper Display e DisplayFor
Validare l'input dell'utente
Gli attributi di validazione
La classe ValidationAttribute
L'attributo Required
L'attributo Range
L'attributo RegularExpression
L'attributo StringLength
L'attributo Remote

Altri tipi di validazione
Applicare la validazione sulla view
Unobtrusive validation
Referenziare le librerie JavaScript di validazione
Gli helper di validazione
Il tag helper Validation Summary
Personalizzare la validazione
Creare un attributo di validazione
Aggiungere la validazione lato client
Validazione personalizzata del modello
Gestire gli errori nella action
Il model binder
Conclusioni
Capitolo 8 - La gestione dello stato
Come funziona una richiesta HTTP?
Scenari di gestione dello stato
Lo stato con i campi hidden
Lo stato attraverso i cookie
Gestione dello stato nella sessione
Passare valori tramite querystring
Gestione dei dati temporanei con TempData
Mantenere dati di una richiesta: HttpContext.Items
Utilizzare la cache
Local cache
Gestione avanzata della local cache

Distributed cache

HTML caching

Response caching

Conclusioni

Capitolo 9 - Accedere ai dati con ADO.NET

Architettura di ADO.NET

Il namespace System.Data.Common

Data provider

Il namespace System.Data.SqlTypes

Il namespace System.Data

Lavorare con ADO.NET

Stabilire una connessione

Esecuzione di un comando

Scrivere dati in transazione

Lettura del risultato di una query

Modalità disconnessa in ADO.NET

Ricercare dati in un DataSet

Conclusioni

Capitolo 10 - Accedere ai dati con Entity Framework Core

Cosa è un O/RM

Mappare il modello a oggetti sul database

Disegnare le classi

Creare il contesto

Mapping tramite convenzioni
Mapping tramite API
Mapping tramite data annotation
Generare automaticamente le classi dal database
Configurare Entity Framework Core con ASP.NET

Recuperare i dati dal database

Ottimizzare il fetching
Lazy loading
Querying avanzato

Salvare i dati sul database

Persistere un nuovo oggetto
Persistere le modifiche a un oggetto
Cancellare un oggetto dal database
Manipolare gli oggetti nel change tracker

Funzionalità aggiuntive di Entity Framework Core

Conclusioni

Capitolo 11 - Sviluppare servizi RESTful con ASP.NET Core WebAPI

I servizi RESTful

Implementare un endpoint per le API con ASP.NET Core

Status code e gestione degli errori

Formatter e content negotiation

Lavorare con le collection

Eseguire aggiornamenti

Hypermedia As The Engine Of Application State (HATEOAS)

Conclusioni

Capitolo 12 - Sviluppare servizi RESTful avanzati con ASP.NET Core WebAPI, WebHook e SignalR

Documentazione con Swagger

Gestione delle versioni

Gestire endpoint basati su WebHook

Comunicazione in tempo reale con SignalR

Conclusioni

Capitolo 13 - Gestione e diagnostica degli errori

La gestione degli errori a runtime

Gli errori durante lo sviluppo

Gli errori per l'utente

Le pagine per gli status code

Tracciare gli eventi

Configurare il logging

Un provider per il logging personalizzato

L'integrazione con Azure App Service

Conclusioni

Capitolo 14 - Estendere ASP.NET Core

Personalizzare l'host di esecuzione

Iniettare l'utilizzo di una libreria
Configurazione separata dell'app
Avviare servizi in background

Creare librerie con view Razor

Caricare a runtime parti dell'applicazione

I middleware e il contesto HTTP

Creare un middleware personalizzato

Creare un middleware come componente

Concetti avanzati sui middleware

Applicare filtri ai controller MVC

Creare un filtro personalizzato

Utilizzare la dependency injection con i filtri

Conclusioni

Capitolo 15 - Model binder e tag helper personalizzati

Il model binding

Creazione di un model binder personalizzato

Model binding di interfacce

Modifica dei metadati

Sviluppare un tag helper personalizzato

Esporre proprietà sul tag helper
Generare HTML nei tag helper

Conclusioni

Capitolo 16 - Supporto alla globalizzazione e internazionalizzazione

I requisiti minimi

Internazionalizzazione, globalizzazione e localizzazione

Lavorare con le culture

Utilizzare i file di risorse

La localizzazione dei contenuti

Il localization middleware

La localizzazione delle view

Le data annotation

Conclusioni

Capitolo 17 - Autenticazione, autorizzazione e identità

Accedere a un'applicazione

- La fase di login
- Un'identità formata da claim
- La fase di autenticazione
- La fase di autorizzazione

Scegliere un sistema di membership

- Membership con ASP.NET Core Identity
- Membership con Azure Active Directory B2C

Configurare l'autenticazione: account locali

- Autenticazione su cookie con ASP.NET Core Identity
- Autenticazione su cookie senza ASP.NET Core Identity

Autenticazione su token JWT

Configurare l'autenticazione: identità federata

Login con i social network

Login su provider esterni con OAuth 2.0 e OpenID Connect

Autenticazione Windows

Autenticazione con Azure Active Directory

Autorizzazione in ASP.NET Core MVC e Web API

L'attributo AuthorizeAttribute

L'attributo AllowAnonymousAttribute

Autorizzazione in base al ruolo

Usare le policy per criteri avanzati di autorizzazione

Policy con logica di autorizzazione personalizzata

Autorizzazione in base all'authentication scheme

Autorizzazione on demand con IAutorizationService

Conclusioni

Capitolo 18 - Sicurezza nelle applicazioni ASP.NET Core

Personalizzare l'account dell'utente

Aggiungere proprietà personalizzate al profilo dell'utente

Personalizzare la pagina di registrazione

Sicurezza degli account locali

Criteri di sicurezza della password

Chiedere la conferma di registrazione

Reimpostazione della password

Two-factor authentication

Protezione dai tentativi di accesso brute force

Sign-out remoto

Conformità con le normative

Controllo dei dati da parte dell'utente per il GDPR

Rispetto della normativa europea sui cookie

Gestire gli utenti con ASP.NET Core Identity

Creare un utente programmaticamente

Interrogare il database degli utenti locali

Assegnare claim e modificare i dati di login dell'utente

Proteggersi dagli attacchi

Proteggere l'applicazione con un certificato SSL

Forzare il traffico su HTTPS

Cross Site Scripting (XSS)

Cross Site Request Forgery (CSRF o XSRF)

Open Redirect

Cross-Origin Resource Sharing (CORS)

Configurare l'applicazione in maniera sicura

User secrets

Azure Key Vault

Proteggere i dati sensibili

Usare Data Protection API per cifrare dati

Gestire le chiavi crittografiche di Data Protection API

Conclusioni

Capitolo 19 - Utilizzare JavaScript in applicazioni ASP.NET

Gestire le librerie JavaScript

jQuery

Effettuare ricerche nel DOM

Manipolare gli oggetti

Gestire gli eventi

Effettuare chiamate AJAX

jQueryUI

Bootstrap

Vue.js

Vue.js e ASP.NET

Conclusioni

Capitolo 20 - Gli strumenti di Visual Studio per lo sviluppo web

Bundling e minification

Minification

Bundling

Integrazione con ASP.NET Core

Link e Script tag helper

Utilizzare NPM

Utilizzare Gulp

Utilizzare Grunt

Conclusioni

Capitolo 21 - Single Page Application e ASP.NET Core

Introduzione alle Single Page Application

Angular

ReactJS

Conclusioni

Capitolo 22 - Configurare e pubblicare un'applicazione ASP.NET Core

Modificare il comportamento secondo l'ambiente di destinazione

Pubblicazione e hosting

- Configurazione di NGINX
- Configurazione di Apache HTTP Server
- Configurazione di IIS

Pubblicazione con Microsoft Azure

Pubblicazione con Docker

Conclusioni

Informazioni sul Libro

www.hoeplieditore.it/5253-0

www.hoeplieditore.it/8525-5

Il testo contenuto nelle note è scritto in questo formato. Le note contengono informazioni aggiuntive relativamente a un argomento o ad aspetti particolari ai quali vogliamo dare una certa rilevanza.

```
Codice
Codice importante, su cui si vuole porre l'accento
Altro codice
```

namespace classi proprietà metodi eventi
questo stile

questo

www.hoeplieditore.it/7290-3
http://books.aspitalia.com/ASP.NET-Core/

http://code.visualstudio.com/

http://www.microsoft.com/express/sql

http://www.hoeplieditore.it/

http://forum.aspitalia.com/

ASPItalia.com Network

❑

❑

- ❑
- ❑
- ❑

Gli autori

Daniele Bochicchio

E-mail: daniele@aspitalia.com
Twitter: http://twitter.com/dbochicchio

Daniele Bochicchio è Chief Digital Officer in iCubed, uno dei più importanti Microsoft Gold Partner, con sede a Milano (http://www.icubed.it). Aiuta i clienti a costruire applicazioni innovative basate sulle tecnologie web, mobile e cloud ed è responsabile della divisione training, grazie alla sua esperienza di oltre vent'anni nel settore.

È Microsoft Regional Director per l'Italia, un ruolo di advisor riservato a un centinaio di persone nel mondo, che aiutano Microsoft a percepire meglio il mercato. È inoltre Microsoft MVP dal 2002, a riconoscimento del suo impegno nelle community.

Nel 1998 ha ideato e sviluppato ASPItalia.com, una delle storiche community italiane di sviluppatori, di cui coordina ancora le attività e che conta 70 mila utenti. Pubblica codice con licenza Open Source dal 1998, aiutando gli sviluppatori a far girare più facilmente le conoscenze.

Ha all'attivo più di trenta libri pubblicati su tematiche tecniche e potete incontrarlo abitualmente ai più importanti eventi e alle conferenze tecniche italiane e internazionali.

Cristian Civera

E-mail: cristian@aspitalia.com
Twitter: http://twitter.com/CristianCivera

Cristian è Senior Software Architect e opera nello sviluppo di applicazioni web. Le sue competenze si basano sull'intero .NET, di cui si è sempre interessato fin dalla prima versione, sulla piattaforma cloud Microsoft Azure e sui linguaggi e framework client, come Angular e TypeScript. Numerose le sue esperienze lavorative che spaziano dalla domotica, passando dai gestionali fino ad arrivare all'automazione industriale. Contribuisce alla community di ASPItalia.com ed è Microsoft Azure MVP

dal 2004. Partecipa regolarmente a diversi eventi, anche per Microsoft Italia, in qualità di speaker.

Moreno Gentili

E-mail: brightsoul@aspitalia.com
Twitter: http://twitter.com/GentiliMoreno

Moreno è sviluppatore web certificato Microsoft e dal 2002 aiuta le aziende a innovare con software di qualità. Investendo molto nell'auto-formazione, si interessa di svariate discipline, dallo user-centered design al machine learning. Nel corso degli anni si è occupato anche di mentoring, insegnamento e public speaking. Dal 2012 fa parte dello staff di ASPItalia.com e contribuisce alla community scrivendo articoli e interagendo con i membri del forum.

Stefano Mostarda

E-mail: sm@aspitalia.com
Twitter: http://twitter.com/sm15455

Stefano è Software Architect e si occupa di progettazione e sviluppo di applicazioni principalmente Web. Sin dagli inizi della sua carriera ha continuato a seguire le innovazioni tecnologiche aiutando le aziende, in diversi settori chiave come domotica, media e gaming, nel loro processo di automazione.

Dal 2004 è membro dello staff del network **ASPItalia.com** e Content Manager dei siti **LINQItalia.com** e HTML5Italia.com. È MVP nella categoria Visual Studio and Development Technologies ed è autore di diversi libri di questa collana e di libri in lingua inglese, sempre dedicati allo sviluppo .NET, oltre che speaker nelle maggiori conferenze italiane.

Matteo Tumiati

E-mail: matteo@aspitalia.com
Twitter: http://twitter.com/xtumiox

Matteo è Senior Consultant in iCubed nella divisione AppDev e nel suo lavoro di consulenza si occupa di training e sviluppo software, con un focus principale su tecnologie Microsoft come .NET. Aiuta i suoi clienti nella migrazione e nella costruzione di piattaforme cloud-ready con Microsoft Azure e Kubernetes e sfruttando metodologie alla base di DevOps. Dal 2016 è membro dello staff di ASPItalia.com e Content Manager del sito WinRTItalia.com. È inoltre Microsoft MVP nella categoria Windows Development dal 2015 grazie ai numerosi contributi per la community e partecipazioni come speaker presso le principali conferenze nazionali.

Ringraziamenti

Daniele ringrazia Noemi, Alessio e Matteo e tutta la sua famiglia, il team editoriale Hoepli e i coautori, con i quali la collaborazione è sempre più ricca di soddisfazioni. Un ringraziamento va anche ai suoi soci e colleghi in iCubed, che gli lasciano il tempo di dedicarsi a queste attività.

Cristian ringrazia Chiara, Sonia e Arianna. La passione per la tecnologia e la famiglia richiedono tempo e fortunatamente riesce a conciliare entrambe le cose.

Moreno ringrazia Silvia per l'infinita pazienza, per i disegni a mano libera e gli stuzzichini caldi. Un grazie speciale allo staff di ASPItalia.com per averlo coinvolto nella prima esperienza da autore di libri <3.

Stefano ringrazia tutta la sua famiglia, i suoi amici. Grazie in maniera speciale agli altri autori per aver preso parte a quest'avventura. Aho!

Matteo desidera ringraziare la sua famiglia e i suoi amici, per il supporto e la pazienza, gli altri autori, lo staff di ASPItalia.com e in particolare Daniele per la stima e le numerose opportunità.

Il team tiene particolarmente a ringraziare la community di ASPItalia.com Network, a cui anche quest'ultimo libro è, come sempre, idealmente dedicato! Gli autori ringraziano tutte le persone che in Microsoft Corp e Microsoft Italia li hanno supportati durante questi mesi, nell'attesa dell'uscita di Visual Studio 2012.

.NET Core è un reboot di .NET. .NET Framework, infatti, non consente di re-ingegnerizzare tutti i pezzi di cui è composto per garantirne una portabilità cross-platform. Piuttosto che creare versioni di .NET Framework incompatibili tra loro, Microsoft ha preferito differenziarle totalmente, a partire dal nome stesso.

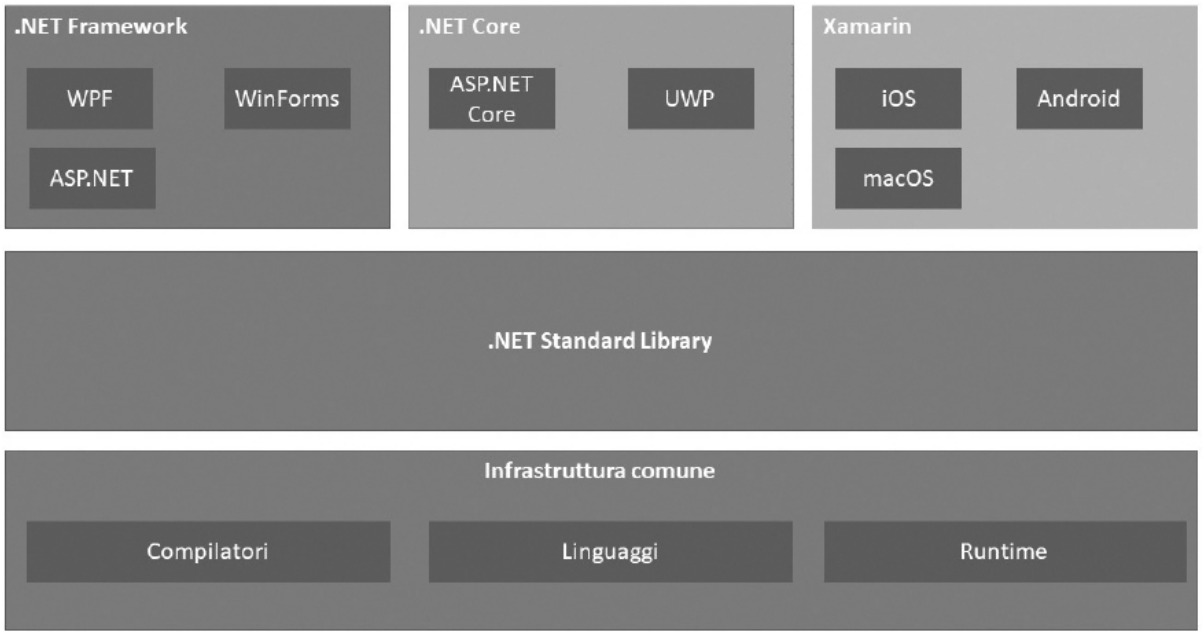
Generalmente, nel corso di questo libro facciamo riferimento in maniera generica a Visual Studio, parlando della versione 2017 Update 15.7. Molti dei concetti e delle funzionalità sono validi anche per tutte le versioni successive. Al momento di scrivere questo libro, Visual Studio 2019 è stato solo annunciato.

http://github.com/dotnet/

- ❑
- ❑
- ❑
- ❑
- ❑
- ❑
- ❑
- ❑
- ❑
- ❑

Creare applicazioni per .NET Core

-
-


.NET Framework
WPF
WinForms
ASP.NET
.NET Core
ASP.NET Core
UWP
Xamarin
iOS
Android
macOS
.NET Standard Library
Infrastruttura comune
Compilatori
Linguaggi
Runtime

- ❑
- ❑
- ❑
- ❑
- ❑
- ❑
- ❑

http://aspit.co/bo4

NuGet è il package manager di riferimento per il mondo .NET. Consente di referenziare facilmente dipendenze attraverso il suo repository centralizzato. Maggiori informazioni sono disponibili su http://www.nuget.org/.

ASP.NET Core e .NET Framework

http://aspit.co/bo5

`NETStandard.Library`

I metapackage sono una convenzione di NuGet per raggruppare logicamente una serie di package che ha senso stiano insieme. Come vedremo, anche ASP.NET Core è distribuito come metapackage ma resta possibile referenziare direttamente, se necessario, anche i singoli package.

`netstandard`

- ❑
- ❑
- ❑

- ❑
- ❑
- ❑
- ❑
- ❑
- ❑

http://www.visualstudio.com

http://aspit.co/bk

IDE o editor?

http://aspit.co/bkc

❑

❑

.NET Downloads Learn Architecture Docs Customers Community Support

Other Windows Downloads

.NET Core	Current (v2.0.3)		Long Term Support (v1.1.5)	
x64 Installer (.exe)	SDK	Runtime	SDK	Runtime
x86 Installer (.exe)	SDK	Runtime	SDK	Runtime
x64 Binaries (.zip)	SDK	Runtime	SDK	Runtime
x86 Binaries (.zip)	SDK	Runtime	SDK	Runtime
Windows Server Hosting (.exe)	N/A	Runtime	N/A	Runtime
Checksums	SDK	Runtime	SDK	Runtime

Visit the download archive for more versions.

❑

❑

Windows	C:\Program Files\dotnet
Linux	/usr/bin/dotnet ~/.dotnet
macOS	/usr/local/share/dotnet

Guida all'utilizzo del comando dotnet

`dotnet`

`--help`

`-h`

```
dotnet --help
```

dotnet

new	
restore	
run	
build	
publish	
test	
pack	
migrate	
clean	
sln	
add	
remove	
list	
nuget	
msbuild	
vstest	

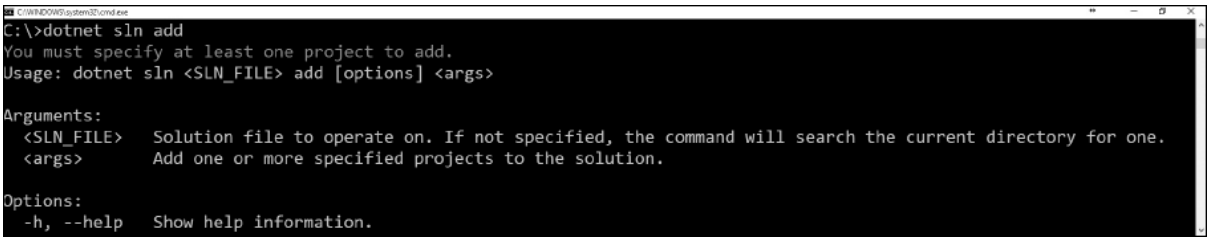
dotnet

dotnet

store	
tool	
buildserver	
dev-certs	
ef	
sql-cache	
watch	
help	

dotnet

--help


C:\WINDOWS\system32\cmd.exe
C:\>dotnet sln add
You must specify at least one project to add.
Usage: dotnet sln <SLN_FILE> add [options] <args>
Arguments:
<SLN_FILE> Solution file to operate on. If not specified, the command will search the current directory for one.
<args> Add one or more specified projects to the solution.
Options:
-h, --help Show help information.

- ❑
- ❑
- ❑

Elencare le SDK installate e determinare quella in uso

dotnet --list-sdks

```
dotnet --list-sdks
```

--version

```
dotnet --version
```

--info

```
dotnet --info
```

```
Prompt dei comandi
C:\>dotnet --info
.NET Core SDK (che rispecchia un qualsiasi file global.json):
 Version:   2.1.300
 Commit:    adab45bf0c

Ambiente di runtime:
 OS Name:     Windows
 OS Version:  10.0.17134
 OS Platform: Windows
 RID:         win10-x64
 Base Path:   C:\Program Files\dotnet\sdk\2.1.300\

Host (useful for support):
  Version: 2.1.0
  Commit:  caa7b7e2ba
```

`dotnet --info`

Cambiare la versione del .NET Core SDK con il global.json

`dotnet new globaljson`

```
{
  "sdk": {
  "version": "2.1.0"
  }
}
```

dotnet --version

dotnet

- ❑
- ❑
- ❑
- ❑
- ❑
- ❑

Scegliere un template di progetto

```
dotnet new
```

dotnet new

web		
mvc		
razor		
webapi		
angular		
react		
reactredux		
classlib		
mstest, xunit		

sln		
globaljson		
nugetconfig		
webconfig		

- mvc razor
- webapi
- angular react reactredux

❑

web

mvc

```
dotnet new mvc
```

--language -lang --output -o

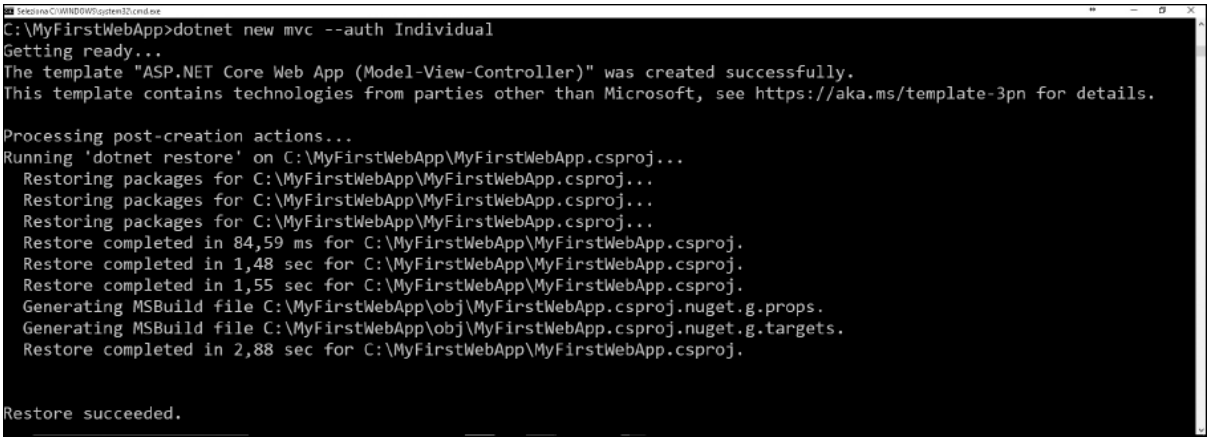
```
dotnet new mvc --language F# --output MyProject
```

--help -h

```
dotnet new mvc --help
```

mvc

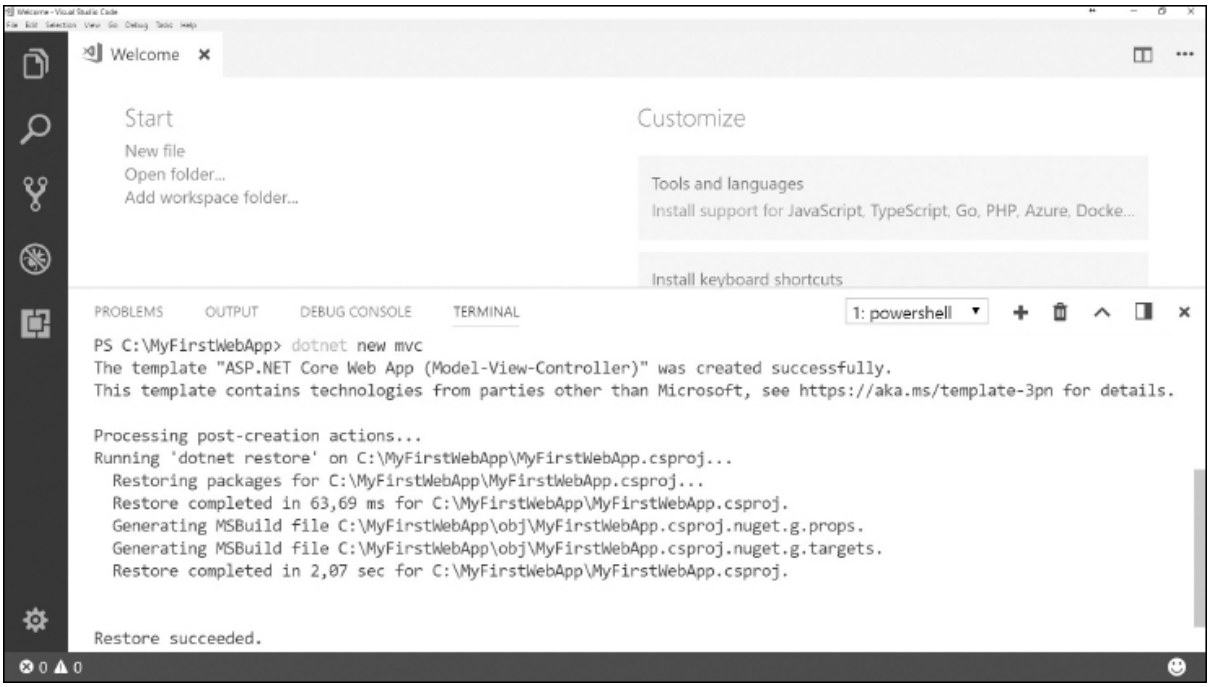
```
dotnet new mvc --auth Individual
```



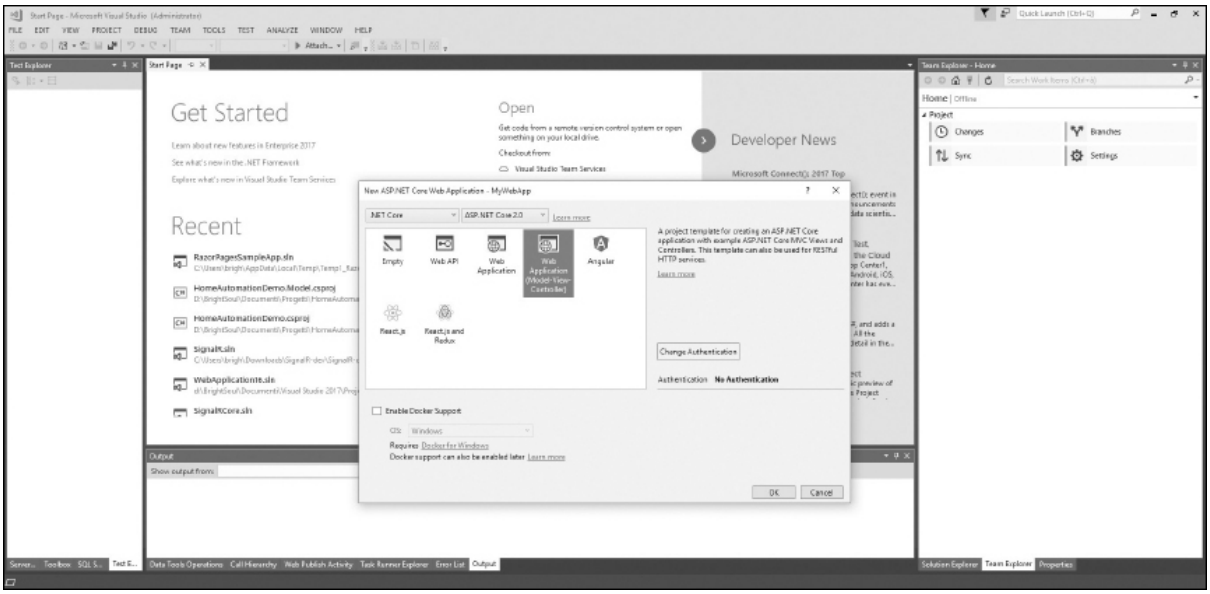
```
C:\MyFirstWebApp>dotnet new mvc --auth Individual
Getting ready...
The template "ASP.NET Core Web App (Model-View-Controller)" was created successfully.
This template contains technologies from parties other than Microsoft, see https://aka.ms/template-3pn for details.

Processing post-creation actions...
Running 'dotnet restore' on C:\MyFirstWebApp\MyFirstWebApp.csproj...
  Restoring packages for C:\MyFirstWebApp\MyFirstWebApp.csproj...
  Restoring packages for C:\MyFirstWebApp\MyFirstWebApp.csproj...
  Restoring packages for C:\MyFirstWebApp\MyFirstWebApp.csproj...
  Restore completed in 84,59 ms for C:\MyFirstWebApp\MyFirstWebApp.csproj.
  Restore completed in 1,48 sec for C:\MyFirstWebApp\MyFirstWebApp.csproj.
  Restore completed in 1,55 sec for C:\MyFirstWebApp\MyFirstWebApp.csproj.
  Generating MSBuild file C:\MyFirstWebApp\obj\MyFirstWebApp.csproj.nuget.g.props.
  Generating MSBuild file C:\MyFirstWebApp\obj\MyFirstWebApp.csproj.nuget.g.targets.
  Restore completed in 2,88 sec for C:\MyFirstWebApp\MyFirstWebApp.csproj.


Restore succeeded.
```

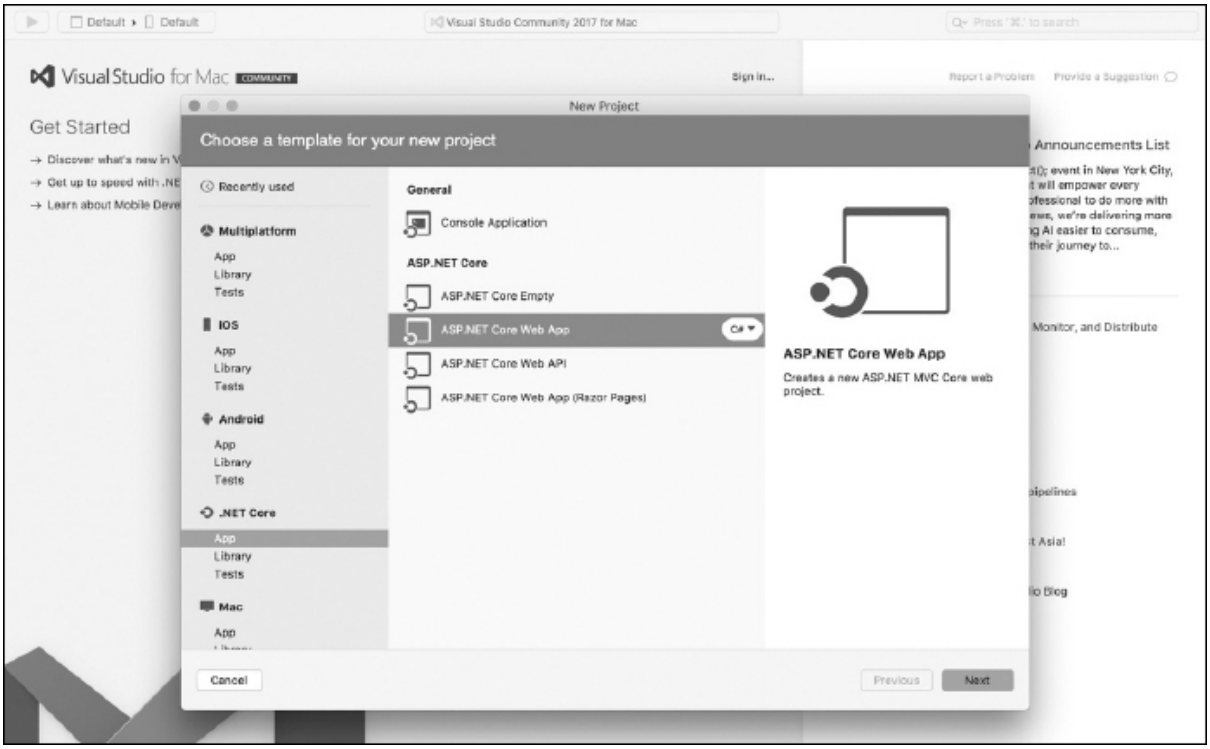
dotnet new

Creare un progetto dagli ambienti di sviluppo



File > New > Project ASP.NET Core Web Application


Get Started
Open
Developer News
Recent
New ASP.NET Core Web Application - MyWebApp
Empty
Web API
Web Application
Web Application (Model-View-Controller)
Angular
React.js
React.js and Redux
Change Authentication
Authentication No Authentication
Enable Docker Support
OK
Cancel


Visual Studio for Mac
COMMUNITY
Sign in...
Get Started
New Project
Choose a template for your new project
Recently used
Multiplatform
App
Library
Tests
iOS
App
Library
Tests
Android
App
Library
Tests
.NET Core
App
Library
Tests
Mac
App
General
Console Application
ASP.NET Core
ASP.NET Core Empty
ASP.NET Core Web App
ASP.NET Core Web API
ASP.NET Core Web App (Razor Pages)
ASP.NET Core Web App
Creates a new ASP.NET MVC Core web project.
Cancel
Previous
Next

Installare altri template di progetto

http://aspit.co/bkd

```
dotnet new -i "Popov1024.HttpApi.Template.CSharp::*"
```

```
dotnet new httpapi
```

dotnet build

--configuration -c

-

-

```
dotnet build --configuration Release
```

dotnet build

Startup.cs

```
C:\MyFirstWebApp>dotnet build
Microsoft (R) Build Engine version 15.4.8.50001 for .NET Core
Copyright (C) Microsoft Corporation. All rights reserved.

Startup.cs(36,41): error CS1010: Newline in constant [C:\MyFirstWebApp\MyFirstWebApp.csproj]
Startup.cs(36,55): error CS1026: ) expected [C:\MyFirstWebApp\MyFirstWebApp.csproj]
Startup.cs(36,55): error CS1002: ; expected [C:\MyFirstWebApp\MyFirstWebApp.csproj]

Build FAILED.

Startup.cs(36,41): error CS1010: Newline in constant [C:\MyFirstWebApp\MyFirstWebApp.csproj]
Startup.cs(36,55): error CS1026: ) expected [C:\MyFirstWebApp\MyFirstWebApp.csproj]
Startup.cs(36,55): error CS1002: ; expected [C:\MyFirstWebApp\MyFirstWebApp.csproj]
    0 Warning(s)
    3 Error(s)

Time Elapsed 00:00:06.75
```

dotnet run

```
dotnet run -c Release
```

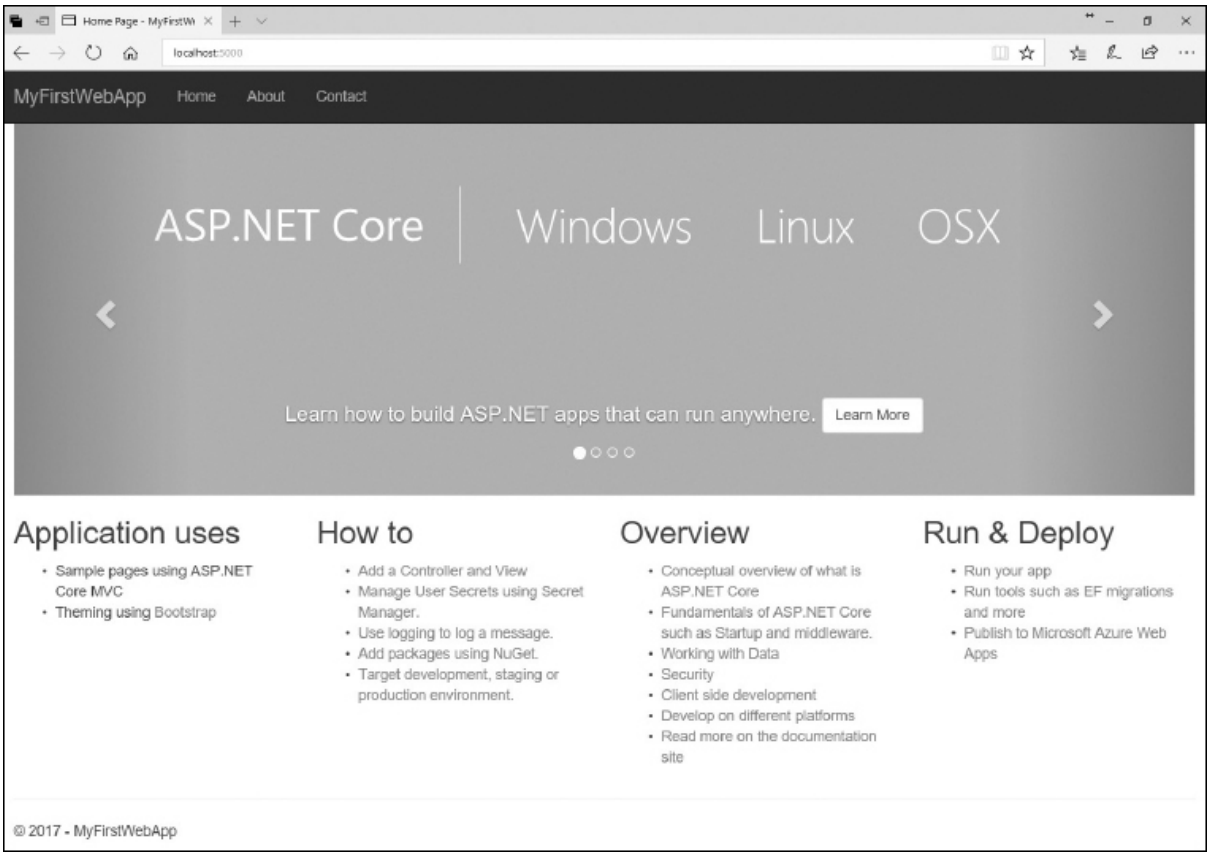
```
C:\MyFirstWebApp>dotnet run
Hosting environment: Production
Content root path: C:\MyFirstWebApp
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
```

dotnet run

http://localhost:5000/

mvc

dotnet new



Le applicazioni ASP.NET Core sono applicazioni console

dotnet run,
chiave=valore
-c
dotnet log

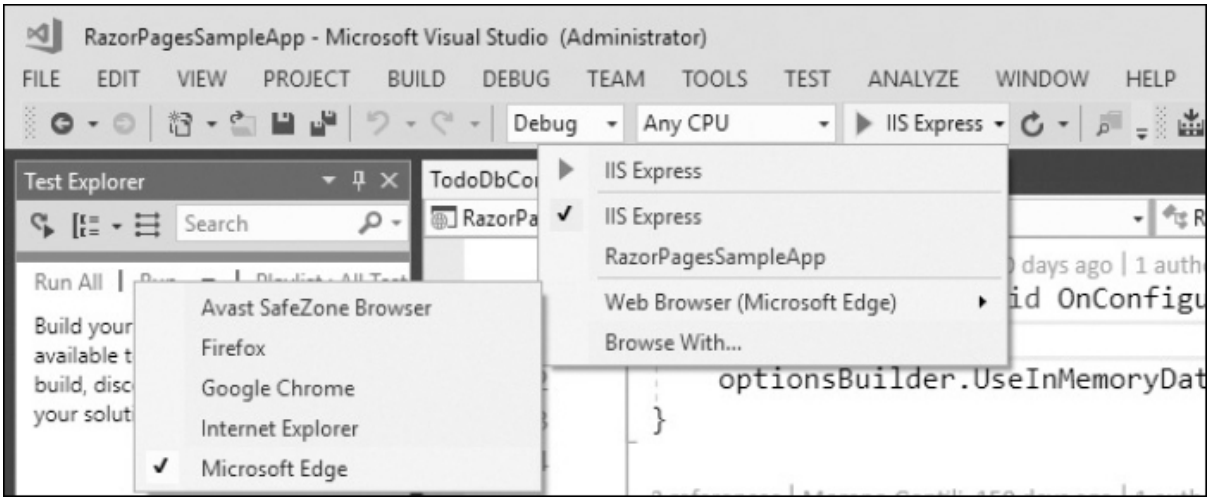
```
dotnet run -c Release log=info
```

- -- /

http://aspit.co/bke

Debug con Visual Studio

File > Open > Project/Solution

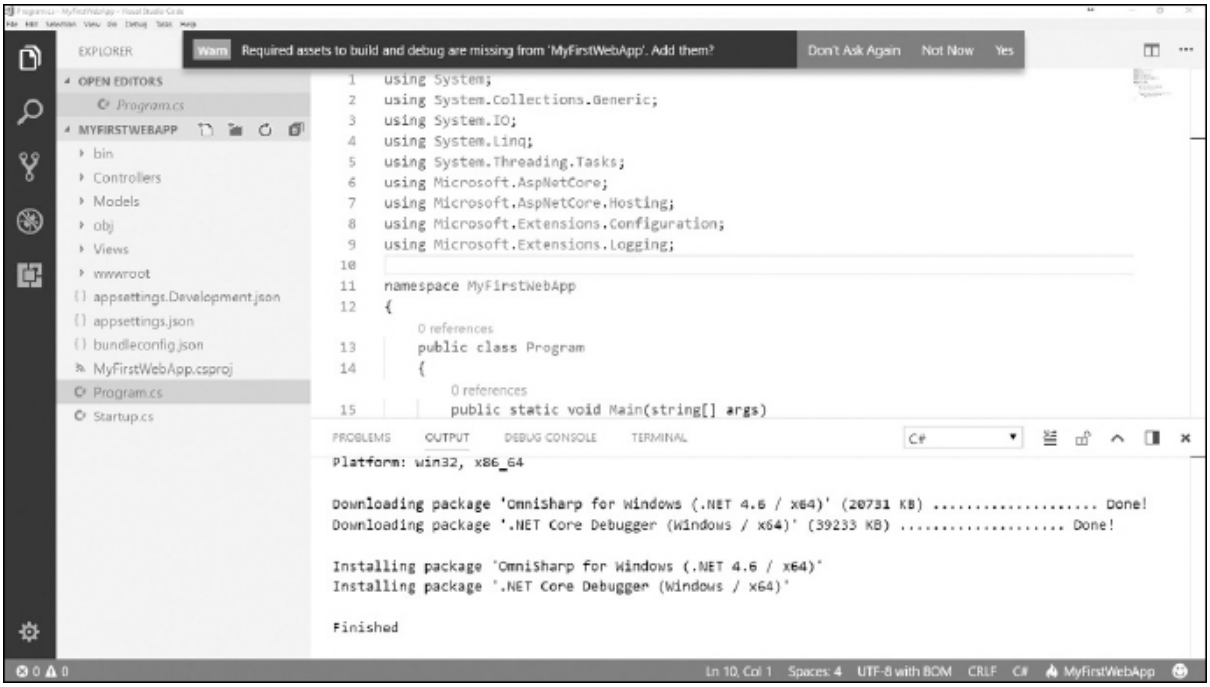


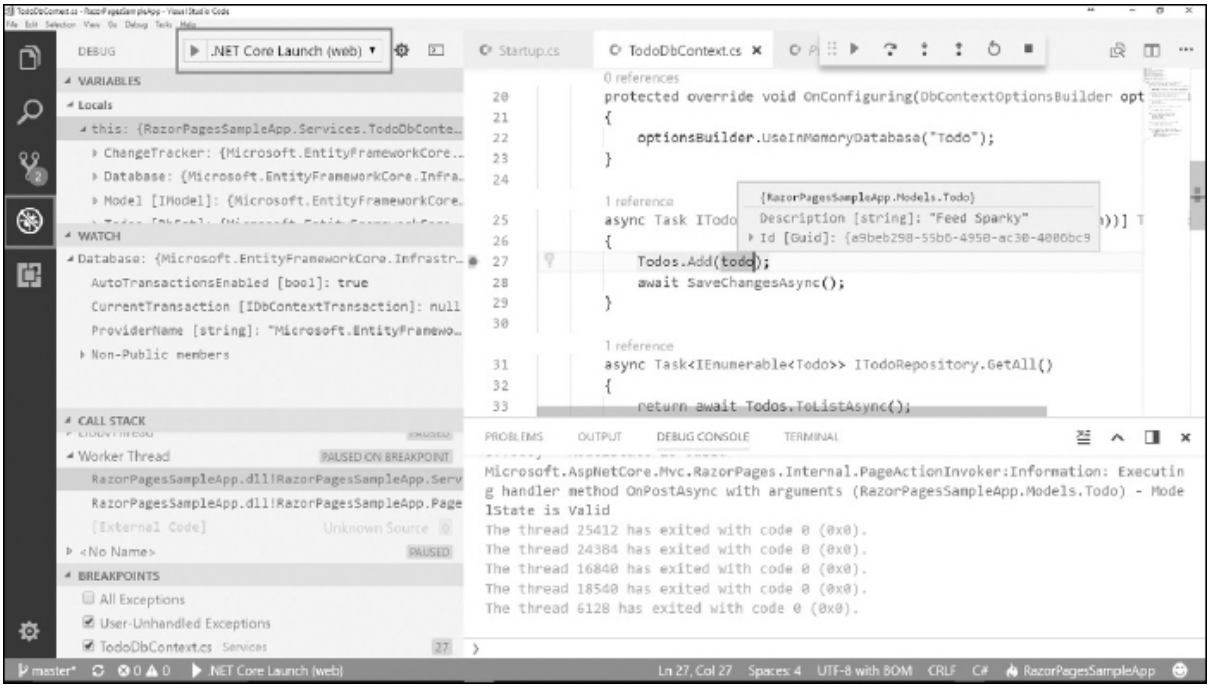

```
25    async Task ITodoRepository.Add([Bind(nameof(Todo.Description))] Todo todo)
26    {
27        Todos.Add(todo);
28        await SaveChangesAsync();
29    }
30
```

```
25    async Task ITodoRepository.Add([Bind(nameof(Todo.Description))] Todo todo)
26    {
27        if (string.IsNullOrEmpty(todo.Description))
28            throw new
29        Todos.Add(todo);
30        await SaveChangesAsync();
```

Debug con Visual Studio Code

File > Open Folder


EXPLORER
Warn
Required assets to build and debug are missing from 'MyFirstWebApp'. Add them?
Don't Ask Again
Not Now
Yes
OPEN EDITORS
Program.cs
MYFIRSTWEBAPP
bin
Controllers
Models
obj
Views
wwwroot
appsettings.Development.json
appsettings.json
bundleconfig.json
MyFirstWebApp.csproj
Program.cs
Startup.cs
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;
namespace MyFirstWebApp
{
0 references
public class Program
{
0 references
public static void Main(string[] args)
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
C#
Platform: win32, x86_64
Downloading package 'Omnisharp for Windows (.NET 4.6 / x64)' (20731 KB) ..... Done!
Downloading package '.NET Core Debugger (Windows / x64)' (39233 KB) ..... Done!
Installing package 'OmniSharp for Windows (.NET 4.6 / x64)'
Installing package '.NET Core Debugger (Windows / x64)'
Finished
Ln 10, Col 1
Spaces: 4
UTF-8 with BOM
CRLF
C#
MyFirstWebApp

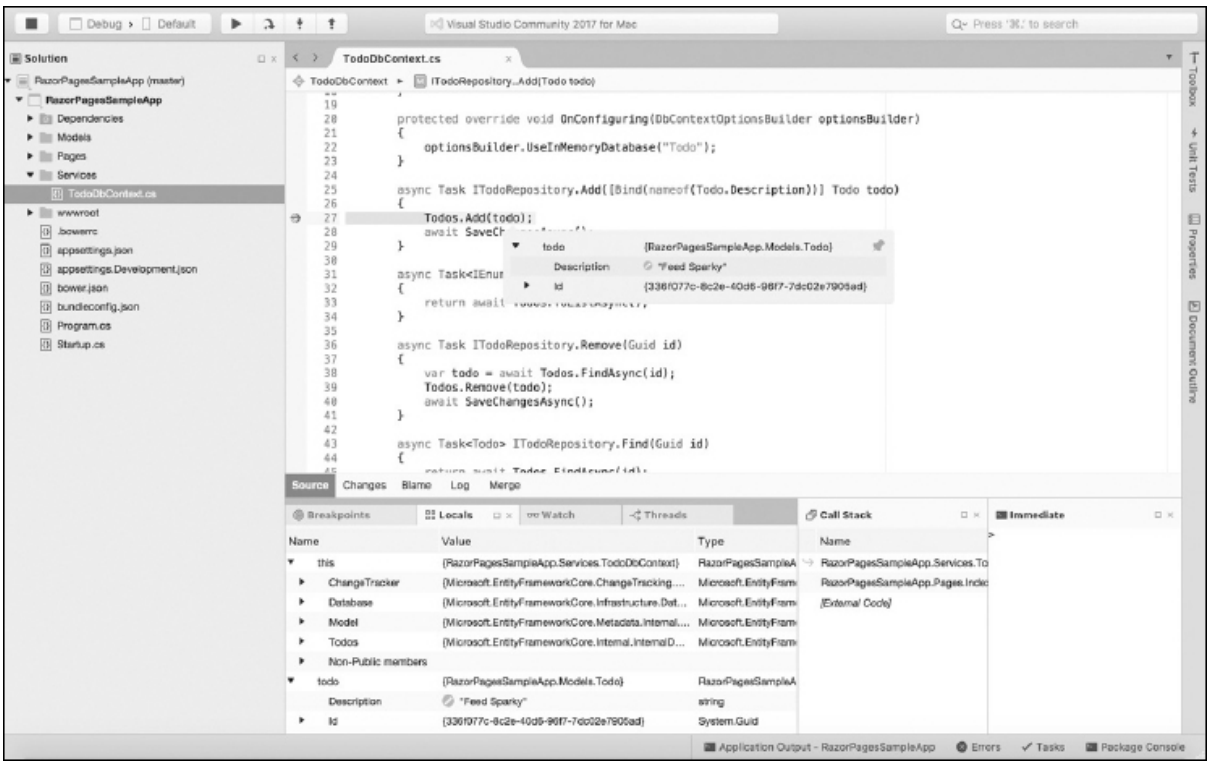


Debug con Visual Studio for Mac

❑

❑

❑


Debug › Default
Visual Studio Community 2017 for Mac
Solution
RazorPagesSampleApp (master)
RazorPagesSampleApp
Dependencies
Models
Pages
Services
TodoDbContext.cs
wwwroot
.bowerrc
appsettings.json
appsettings.Development.json
bower.json
bundleconfig.json
Program.cs
Startup.cs
TodoDbContext.cs
TodoDbContext
ITodoRepository...Add(Todo todo)
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
optionsBuilder.UseInMemoryDatabase("Todo");
}
async Task ITodoRepository.Add([Bind(nameof(Todo.Description))] Todo todo)
{
Todos.Add(todo);
}
async Task ITodoRepository.Remove(Guid id)
{
var todo = await Todos.FindAsync(id);
Todos.Remove(todo);
await SaveChangesAsync();
}
async Task<Todo> ITodoRepository.Find(Guid id)
{
todo {RazorPagesSampleApp.Models.Todo}
Description "Feed Sparky"
Id {336f077c-8c2e-40d6-96f7-7dc02e7905ad}
Source
Changes
Blame
Log
Merge
Breakpoints
Locals
Watch
Threads
Call Stack
Immediate
Name
Value
Type
this {RazorPagesSampleApp.Services.TodoDbContext}
ChangeTracker
Database
Model
Todos
Non-Public members
todo {RazorPagesSampleApp.Models.Todo}
Description "Feed Sparky"
Id {336f077c-8c2e-40d6-96f7-7dc02e7905ad}
string
System.Guid
[External Code]
Application Output - RazorPagesSampleApp
Errors
Tasks
Package Console
Toolbox
Unit Tests
Properties
Document Outline

Creare la solution

`.sln`

```
dotnet new sln
```

.sln

```
dotnet new mvc --output MyWebApp
dotnet sln add ./MyWebApp/MyWebApp.csproj
```

.csproj

Aggiungere un progetto di tipo Class Library

Prodotto

dotnet new

```
dotnet new classlib --output MyLib
dotnet sln add ./MyLib/MyLib.csproj
dotnet add ./MyWebApp/MyWebApp.csproj reference ./MyLib/MyLib.csproj
```

dotnet add reference

Aggiungere un progetto di unit testing

MSTest XUnit

http://aspit.co/bke

MSTest

```
dotnet new mstest --output MyTest
dotnet sln add ./MyTest/MyTest.csproj
dotnet add ./MyTest/MyTest.csproj reference ./MyLib/MyLib.csproj
```

```
dotnet test ./MyTest/MyTest.csproj
```

Aggiungere pacchetti NuGet al progetto

http://aspit.co/bkg

iTextSharp

```
dotnet add ./MyFirstWebApp/MyFirstWebApp.csproj package iTextSharp
```

iTextSharp

Creare un nostro personale pacchetto NuGet

.csproj

```
dotnet pack ./MyLib/MyLib.csproj
```

- ❑
- ❑
- ❑

Il comando dotnet publish

```
dotnet publish
```

```
--output
```

```
dotnet publish --output Publish
```

Publish

dotnet publish

Cross-compilazione e pubblicazione self-contained

dotnet publish

--runtime ubuntu-x64

```
dotnet publish --runtime ubuntu-x64 --output Publish
```

ubuntu-x64

http://aspit.co/bkh

Publish

--runtime

`--self-contained false`

```
dotnet publish --runtime ubuntu-x64 --self-contained false --output Publish
```

Creare uno store a livello di sistema dei pacchetti NuGet

- dotnet store

--output

- manifest.xml
dotnet publish
--manifest

http://aspit.co/bki

Avviare l'applicazione nella macchina server

dotnet publish

dotnet

```
dotnet MyFirstWebApp.dll
```

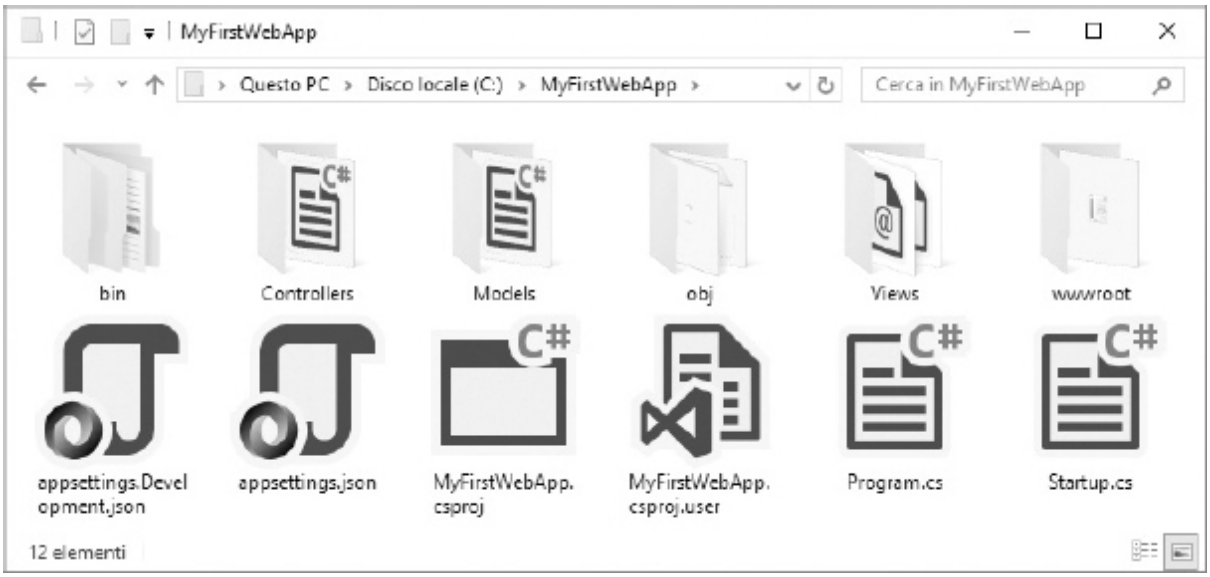
dotnet run run

web.config

❑

❑

❑

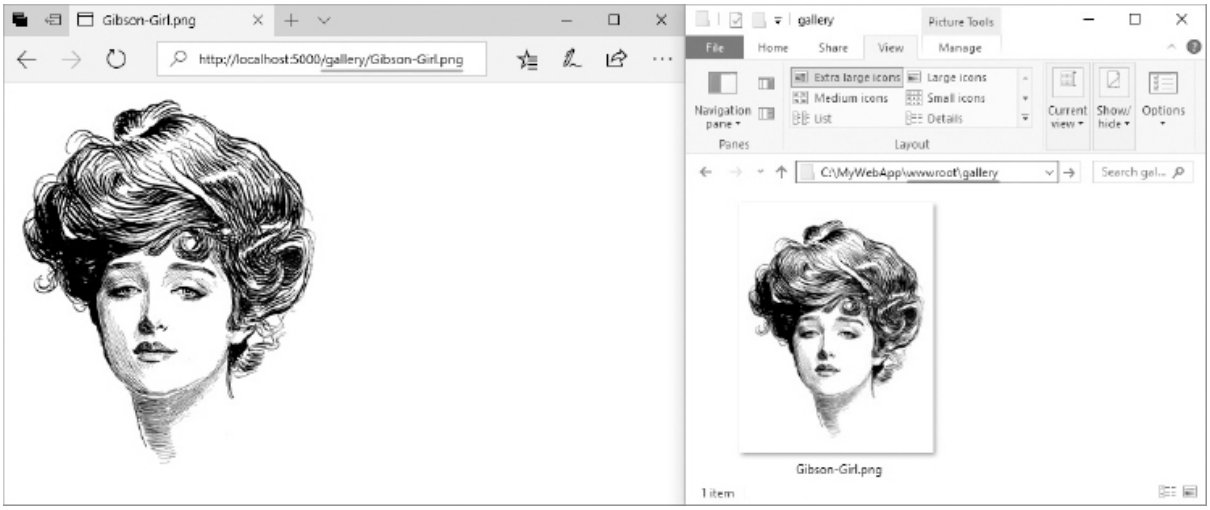


La directory wwwroot

logo.jpg

http://nomehost/logo.jpg

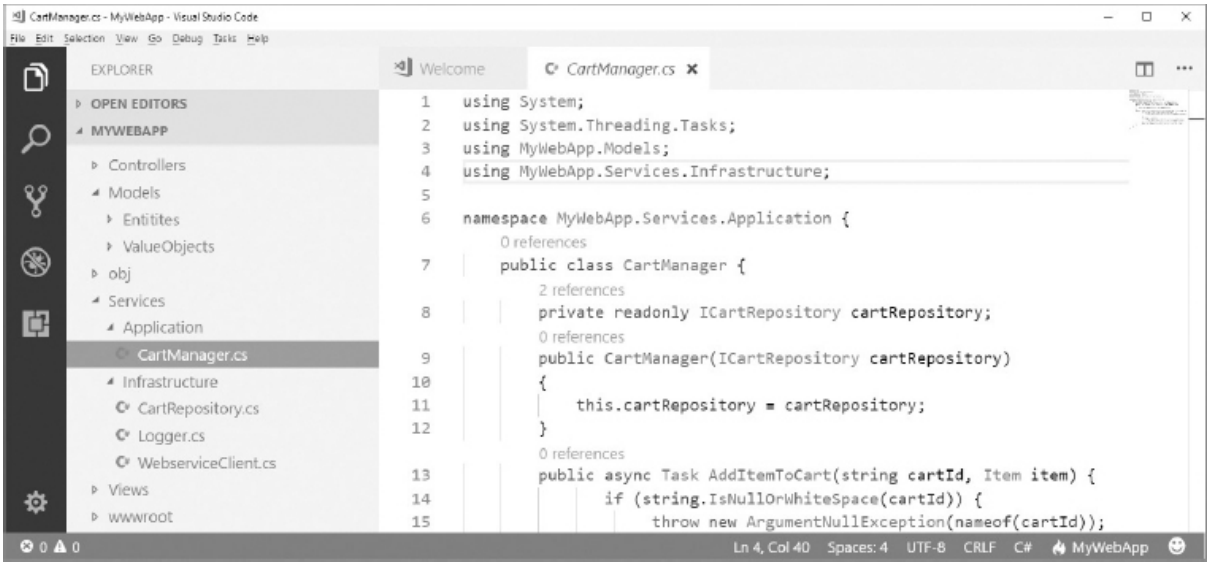
/wwwroot/gallery



gallery

Organizzare il codice in directory

CartManager.cs
/Services/Application
CartManager
MyWebApp.Services.Application



Models Views Controllers

Le directory obje bin

-

-

Debug Release

bin obj

Main

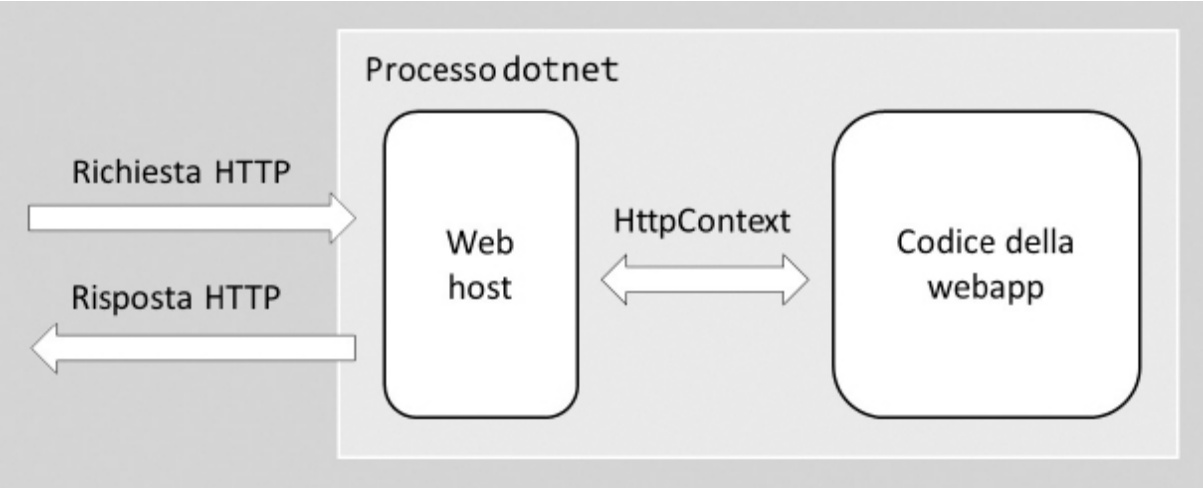
```
public class Program
{
  public static void Main(string[] args)
  {
    BuildWebHost(args).Run();
  }
  public static IWebHost BuildWebHost(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
      .UseStartup<Startup>()
      .Build();
}
```

Main

Program

WebHost.CreateDefaultBuilder

HttpContext



http://aspit.co/bl0

Personalizzare il web host

-
-
-

-
-
-
-

WebHostBuilder
WebHost.CreateDefaultBuilder

Indicare gli endpoint

dotnet run

```
C:\MyWebApp> dotnet run
Hosting environment: Production
Content root path: C:\MyWebApp
```

```
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
```

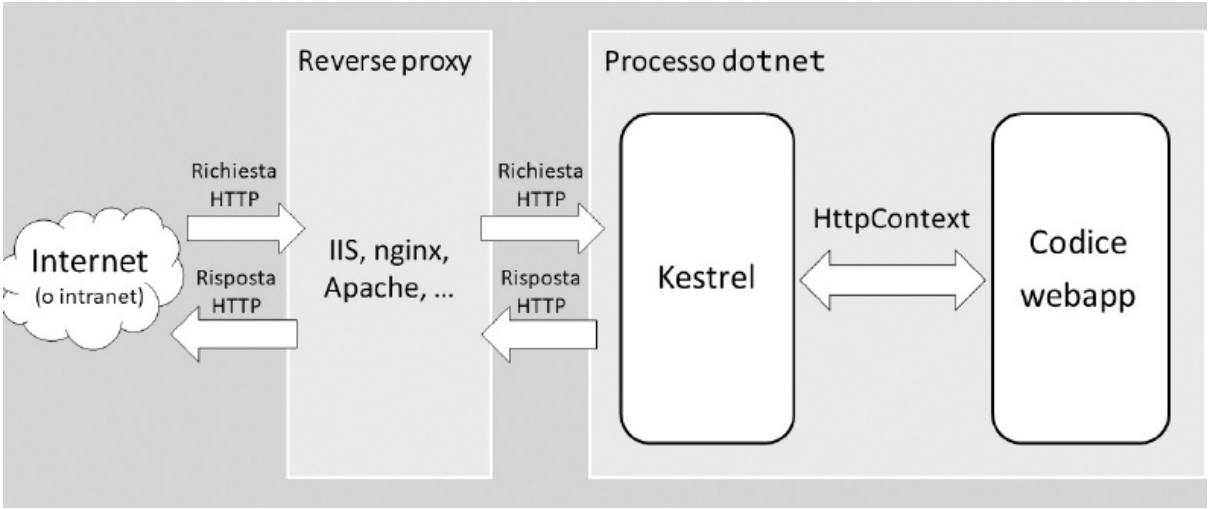
UseUrls

WebHostBuilder

```
public static IWebHost BuildWebHost(string[] args) =>
  WebHost.CreateDefaultBuilder(args)
    .UseUrls("http://*:80", "http://localhost:5000", "http://10.0.0.3:8000")
    .UseStartup<Startup>()
    .Build();
```

```
C:\MyWebApp> dotnet run

Hosting environment: Production
Content root path: C:\MyWebApp
Now listening on: http://[::]:80
Now listening on: http://localhost:5000
Now listening on: http://10.0.0.3:8000
Application started. Press Ctrl+C to shut down.
```

Scegliere un web server: Kestrel o HTTP.sys?

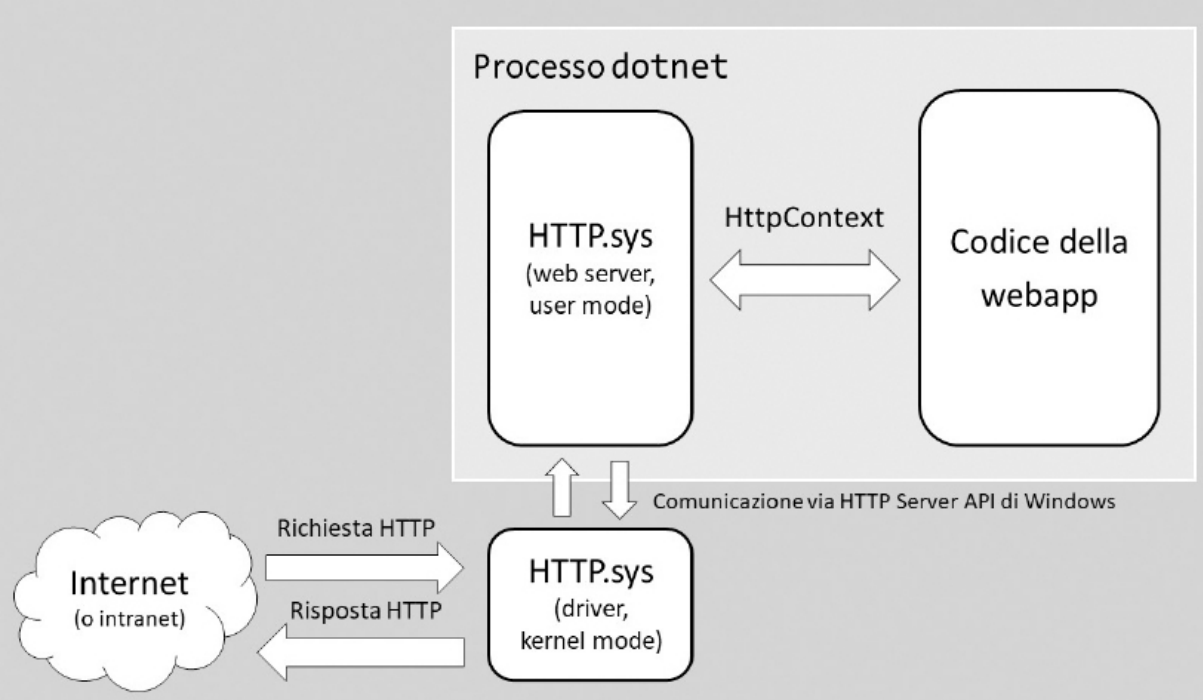


UseKestrel

```
public static IWebHost BuildWebHost(string[] args) =>
  WebHost.CreateDefaultBuilder(args)
    .UseKestrel(options =>
    {
      //Usiamo l´oggetto options per configurare Kestrel
      //Ad esempio, limitiamo la dimensione della richiesta a 10 MB
      options.Limits.MaxRequestBodySize = 10 * 1024 * 1024;
    })
    //Abilitiamo l´integrazione con IIS, che agisce da reverse proxy
    .UseIISIntegration()
    .UseStartup<Startup>()
    .Build();
```

UseIISIntegration

http://aspit.co/bl1



UseHttpSys

```
public static IWebHost BuildWebHost(string[] args) =>
  WebHost.CreateDefaultBuilder(args)
    .UseHttpSys(options => {
      //Abilitamo l´autenticazione con account Windows
      options.Authentication.Schemes =
          AuthenticationSchemes.NTLM |  AuthenticationSchemes.Negotiate;
      options.Authentication.AllowAnonymous = false;
    })
    .UseStartup<Startup>()
    .Build();
```


UseServer

Cambiare il percorso delle directory principali

-

Program Startup

-

UseContentRoot UseWebRoot

```
public static IWebHost BuildWebHost(string[] args) =>
  WebHost.CreateDefaultBuilder(args)
     .UseContentRoot(@"C:\mycontentroot")
     .UseWebRoot("wwwroot2")
     .UseStartup<Startup>()
     .Build();
```

Impostare un environment

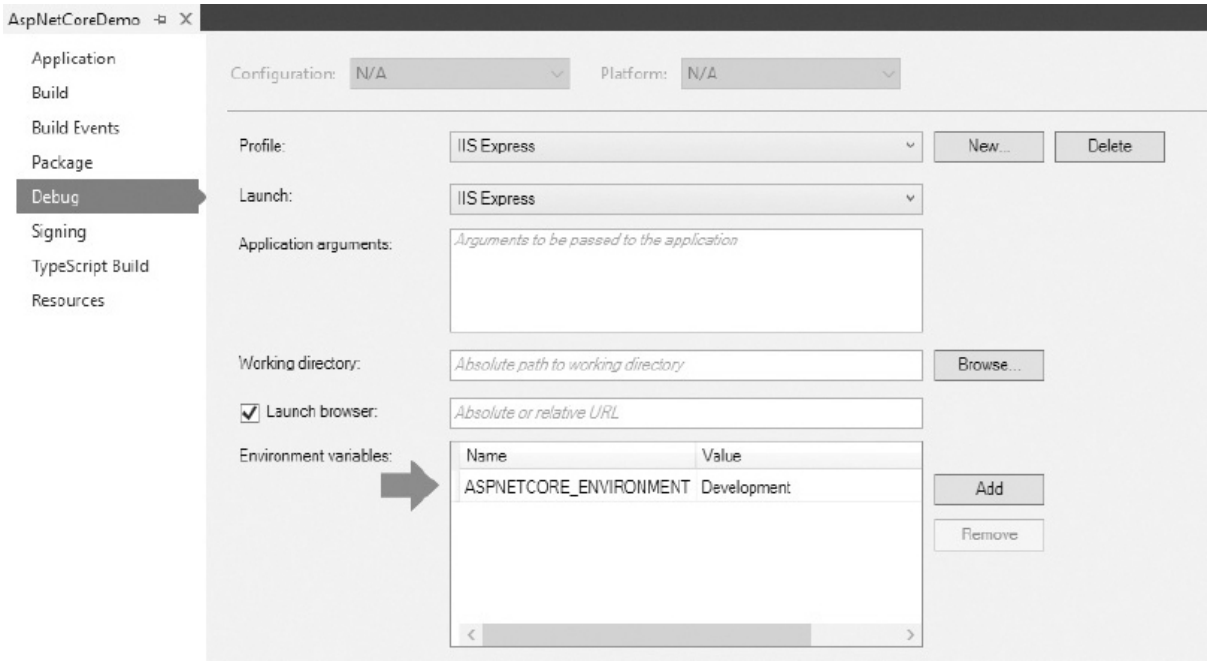
❑

-
-

UseEnvironment

```
public static IWebHost BuildWebHost(string[] args) {
  return WebHost.CreateDefaultBuilder(args)
    //Possiamo scegliere il nome tra le costanti di EnvironmentName
    //oppure fornire una nostra stringa arbitraria (es. ˝Staging2˝)
    .UseEnvironment(EnvironmentName.Development)
    .UseStartup<Startup>()
    .Build();
}
```

ASPNETCORE_ENVIRONMENT



Configurare il web host da fonti esterne

```
Program
ConfigurationBuilder
```

Add

hosting.json ASPNETCORE_

UseConfiguration

```
public static IWebHost BuildWebHost(string[] args) {
  var configuration = new ConfigurationBuilder()
    .SetBasePath(Directory.GetCurrentDirectory())
    .AddJsonFile("hosting.json", optional:true)
    .AddEnvironmentVariables("ASPNETCORE_")
    .Build();
  return WebHost.CreateDefaultBuilder(args)
    .UseStartup<Startup>()
    .UseConfiguration(configuration)
    .Build();
}
```

hosting.json

```
{
  "urls": "http://*:80;http://localhost:5000;http://10.0.0.3:8000",
  "environment": "Development",
  "webRoot": "wwwroot"
}
```

	setx ASPNETCORE_URLS ″http://*:5000″
	echo ″ASPNETCORE_URLS=http://*:5000″>>/etc/environment
	echo ″export ASPNETCORE_URLS=http://*:5000″>>~/.bash_profile

endpoint

http://aspit.co/bl2

Scegliere la classe di configurazione dell'applicazione

UseStartup

```
public static IWebHost BuildWebHost(string[] args) =>
  WebHost.CreateDefaultBuilder(args)
    .UseStartup<Startup>()
    .Build();
```

Startup

Startup

- Configure Startup

- ConfigureServices Startup

```
public class Startup
{
  public void ConfigureServices(IServiceCollection services)
  {
    //Qui aggiungiamo i servizi alla collezione services
  }
  public void Configure(IApplicationBuilder app, IHostingEnvironment env)
  {
    //Qui decidiamo quali middleware usare nella pipeline
  }
}
```

dotnet new web

Configure

Startup

dotnet new mvc

Use

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
```

```
{
  if (env.IsDevelopment())
  {
    //Middleware per la pagina di errore contenente dettagli tecnici
    //(Solo se l´environment è stato impostato su ˝Development˝)
    app.UseDeveloperExceptionPage();
  }
  else
  {
    //Middleware che in caso di errore reindirizza a un determinato url
    //(Solo se l´environment è diverso da ˝Development˝)
    app.UseExceptionHandler(˝/Home/Error˝);
  }
  //Middleware che aggiunge il supporto ai file statici
  app.UseStaticFiles();
  //Middleware di routing di ASP.NET Core MVC
  app.UseMvc(routes =>
  {
    routes.MapRoute(
      name: ˝default˝,
      template: ˝{controller=Home}/{action=Index}/{id?}˝);
  });
}
```

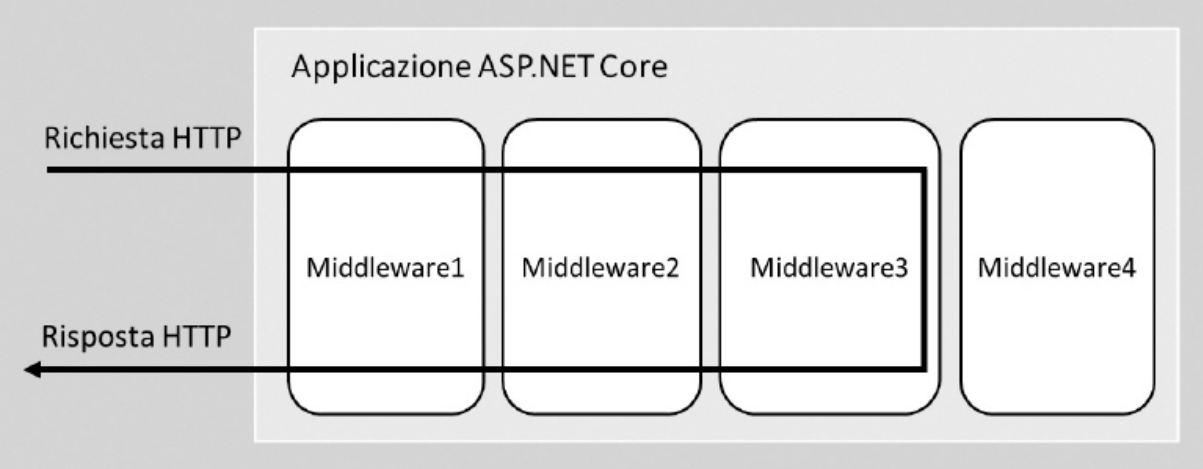
IsDevelopment IHostingEnvironment

Come funzionano i middleware

-

-

❑


Applicazione ASP.NET Core
Richiesta HTTP
Middleware1
Middleware2
Middleware3
Middleware4
Risposta HTTP

❑

❑

❑

Configure

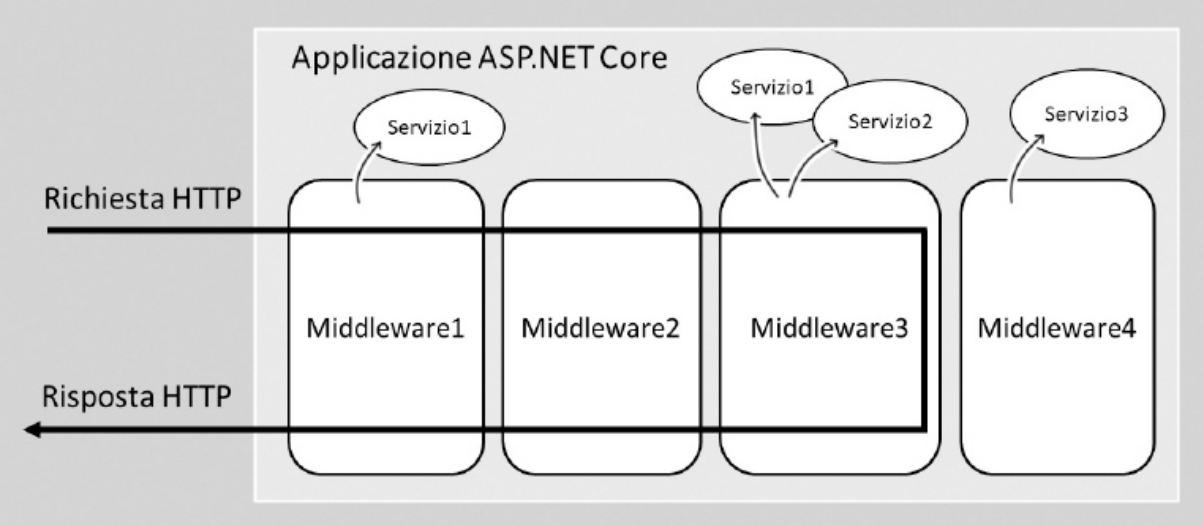
http://aspit.co/bl3

ConfigureServices Startup

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddMvc();
}
```

```
IServiceCollection

Add
```

Cos'è un servizio



appsettings.json

ConfigurationBuilder

Variare la configurazione in base all'environment

IHostingEnvironment

Startup

```
  public class Startup
  {
    public Startup(IHostingEnvironment en)
    {
      //Aggiungiamo le fonti di configurazione desiderate
      var builder = new ConfigurationBuilder()
        .SetBasePath(en.ContentRootPath)
        .AddJsonFile(″appsettings.json″, optional: true, reloadOnChange: true)
        .AddJsonFile($″appsettings.{en.EnvironmentName}.json″, optional: true)
        .AddEnvironmentVariables(″ASPNETCORE_″);
      Configuration = builder.Build();
    }

    //Questa proprietà mantiene un riferimento alla configurazione creata
    public IConfigurationRoot Configuration { get; }

    //I metodi Configure e ConfigureServices sono omessi per brevità
  }
```

appsettings.json

EnvironmentName

appsettings.Development.json

appsettings.json

appsettings.Development.json

{ ″ConnectionStrings″: { ″Default″: ″Server=MainSrv; Database=Db1; User Id=user1; Password=pass1″ }, ″Smtp″: {	{ ″ConnectionStrings″: { ″Default″: ″Server=TestSrv; Database=Test1; User Id=user1; Password=pass1″ } }

```
    "Host": "smtp.example.com",
    "Port": 587,
    "EnableSsl": true,
    "Username": "user1",
    "Password": "pass1",
    "Email": "dev@example.com"
  }
}
```

TestSrv

MainSrv appsettings.Development.json

ASPNETCORE_CONNECTIONSTRINGS:DEFAULT

Leggere la configurazione in maniera fortemente tipizzata

GetSection

```
public void ConfigureServices(IServiceCollection services)
{
  services.Configure<SmtpConfiguration>(Configuration.GetSection("Smtp"));
}
```

services.Configure

SmtpConfiguration

appsettings.json

```
public class SmtpConfiguration
{
  public string Host {get; set;}
  public int Port {get; set;}
  public bool EnableSsl {get; set;}
  public string Username {get; set;}
  public string Password {get; set;}
  public string Email {get; set;}
}
```

IOptionsMonitor<SmtpConfiguration>

```
public class EmailErrorsMiddleware {
  private readonly SmtpConfiguration smtpConf;
  public EmailErrorsMiddleware(IOptionsMonitor<SmtpConfiguration> options)
  {
    //Le impostazioni attuali si trovano nella proprietà CurrentValue
    smtpConf = options.CurrentValue;
  }
}
```

IOptionsMonitor<T>

CurrentValue

IOptions<T>

config

Che fine hanno fatto il web.config e il global.asax?

system.webServer

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <handlers>
      <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModule"
           resourceType="Unspecified" />
    </handlers>
    <aspNetCore processPath="dotnet" arguments=".\MyWebApp.dll"
```

```
                  stdoutLogEnabled="false" stdoutLogFile=".\logs\stdout" />
    </system.webServer>
  </configuration>
```

system.webServer

http://aspit.co/bl4

Configure

ConfigureServices Startup

```
  <Project Sdk="Microsoft.NET.Sdk.Web">
    <PropertyGroup>
      <TargetFramework>netcoreapp2.1</TargetFramework>
    </PropertyGroup>
    <ItemGroup>
      <Folder Include="wwwroot\" />
    </ItemGroup>
```

```
    <ItemGroup>
      <PackageReference Include="Microsoft.AspNetCore.App" Version="2.1.0" />
    </ItemGroup>
  </Project>
```

netcoreapp2.1

Microsoft.AspNetCore.App

Program Startup

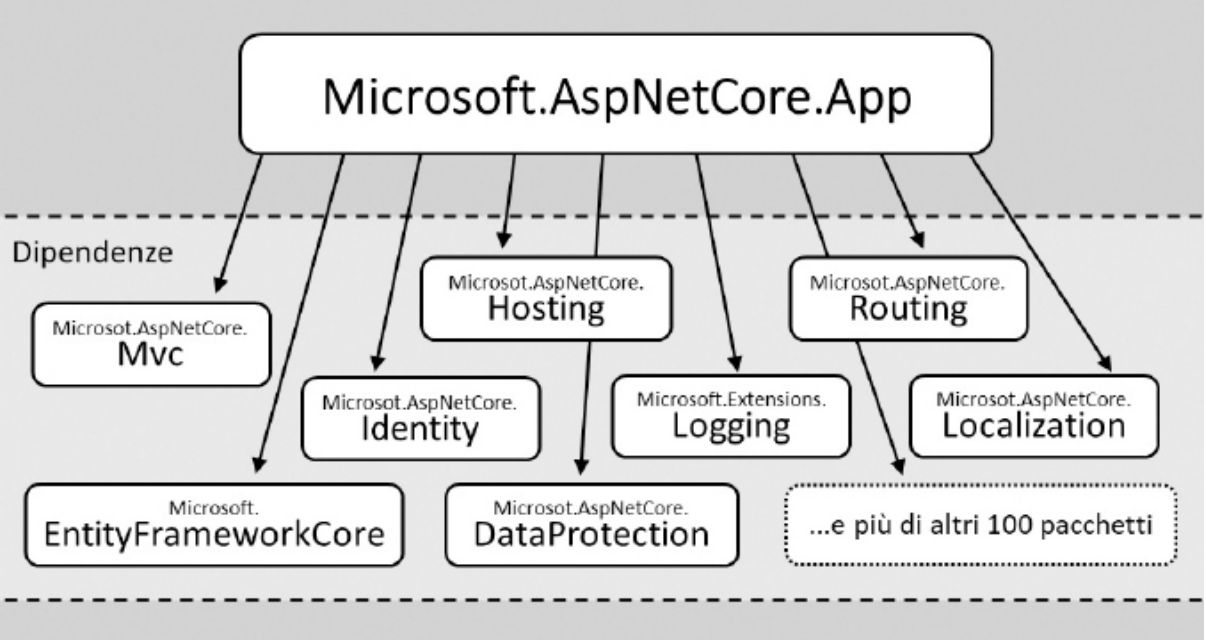
Il metapacchetto Microsoft.AspNetCore.App

Microsoft.AspNetCore.App

Microsoft.AspNetCore.App

Microsoft.AspNetCore.App

http://aspit.co/bnr



Microsoft.AspNetCore.App

Microsoft.Data.Sqlite Microsoft.Extensions.Caching.Redis

Microsoft.AspNetCore.All

Il tool Microsoft.VisualStudio.Web.CodeGeneration.Tools

Microsoft.VisualStudio.Web.CodeGeneration.Tools
aspnet-codegenerator

```
dotnet aspnet-codegenerator
```

Microsoft.VisualStudio.Web.CodeGeneration.Design

4

Primi passi con ASP.NET Core MVC

Se siamo sviluppatori ASP.NET di lungo corso, saremo già abituati ad ASP.NET Web Forms. Tradizionalmente, ASP.NET Web Forms, ha fornito un eccellente livello di astrazione nell'ambito della realizzazione di pagine web, volto a colmare tutti quei limiti che contraddistinguono il protocollo HTTP. Il modello di sviluppo proposto, infatti, è per molti aspetti analogo a quello delle applicazioni client, basato su oggetti che reagiscono alle azioni dell'utente sollevando eventi, e la stessa natura stateless del web diventa assolutamente impercettibile, grazie al ViewState che è in grado di mantenere lo stato del succedersi delle richieste.

Si tratta di un modello di sviluppo che, negli anni, si è dimostrato assolutamente valido per realizzare applicazioni di qualsiasi livello di complessità, ma che di certo non è esente da difetti: il runtime di ASP.NET, l'infrastruttura delle pagine e dei controlli server, infatti, è un insieme monolitico, difficilmente scindibile nelle sue singole componenti e quindi, per esempio, difficilmente testabile tramite unit test, o in cui, come sviluppatori, abbiamo un controllo limitato sull'effettivo markup generato per le singole pagine. Questi limiti e le pressanti richieste da parte della community di sviluppatori hanno portato Microsoft a pensare a una nuova piattaforma per lo sviluppo su web, alternativa ad ASP.NET Web Forms: stiamo parlando di ASP.NET MVC.

Con ASP.NET Core, Microsoft ha deciso che ASP.NET MVC fosse l'unico tra i due in grado di garantire un futuro alla propria piattaforma applicativa: quindi, all'interno di ASP.NET Core troviamo il supporto per una particolare versione di ASP.NET MVC, chiamata ASP.NET Core MVC (o, spesso, ASP.NET MVC Core), che è stata creata a partire da quanto

ASP.NET MVC ha introdotto, ma con numerose e interessanti migliorie che andremo a scoprire nel corso dei prossimi capitoli.

Giunto alla versione 2, ASP.NET Core MVC è un framework decisamente maturo e pronto a essere utilizzato per applicazioni anche complesse, che propone un modello di sviluppo moderno, più aderente al funzionamento del web e basato sul pattern Model-View-Controller, uno dei più noti e collaudati pattern per l'architettura del layer di presentazione.

All'implementazione di MVC all'interno di ASP.NET Core è dedicata buona parte di questo libro, poiché, a meno che non sviluppiamo servizi, è molto probabile che avremo a che fare con questo pezzo dell'architettura di ASP.NET Core. Nel corso di questo capitolo e dei prossimi, cercheremo di mettere in luce le peculiarità di questa implementazione e di illustrare come sfruttarne a fondo l'estrema versatilità. In questo capitolo, in particolare, ne daremo una panoramica introduttiva in modo che possiamo, sin da subito, iniziare a familiarizzare con questo modello. Ne consigliamo una rapida lettura anche a chi ha già una buona dimestichezza con ASP.NET MVC.

Il pattern Model-View-Controller

Tutte le volte che dobbiamo gestire un elevato grado di complessità, la soluzione più adeguata è quella di disegnare un'architettura basata su oggetti più semplici, suddividendo tra essi le diverse responsabilità. Questo concetto si applica anche all'interfaccia utente, tant'è che, in letteratura, sono stati formalizzati diversi pattern secondo cui strutturare questo particolare strato applicativo. Tra essi, uno dei più diffusi e collaudati, negli ultimi anni, è il **Model-View-Controller** (MVC). Formulato per la prima volta intorno alla fine degli anni '70, è oggi alla base di numerose tecnologie di sviluppo per il web, tra cui ricordiamo Java Server Pages, Ruby on Rails e, ovviamente, ASP.NET MVC. Lo schema concettuale è rappresentato nella Figura 4.1.

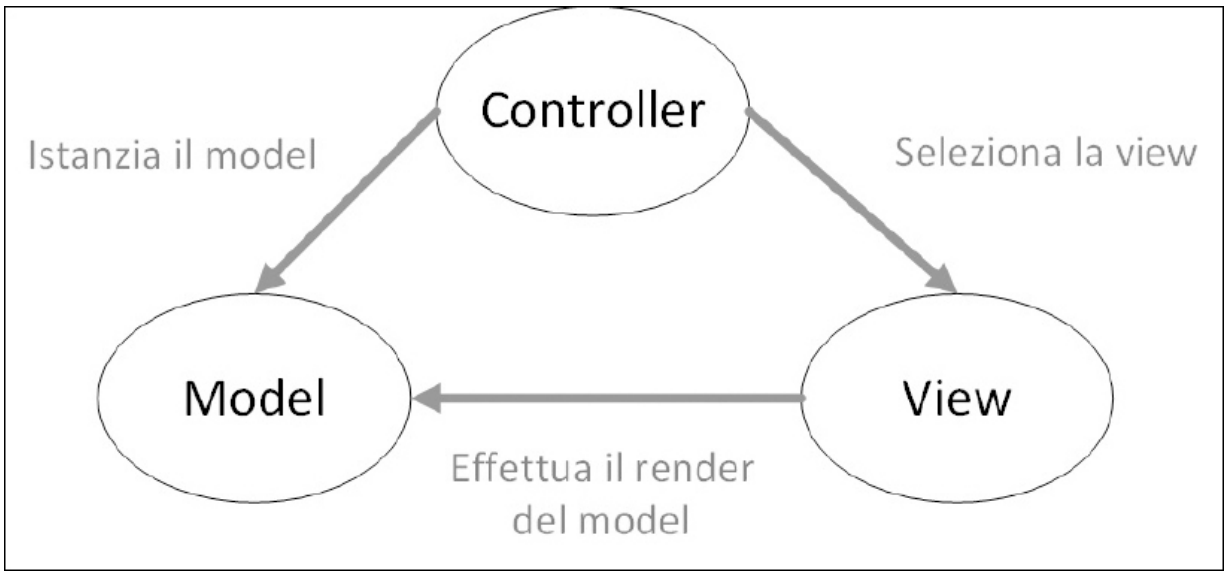


Figura 4.1 – Schema concettuale del pattern Model-View-Controller.

In esso distinguiamo i tre componenti che seguono.

- Il **Model** rappresenta il dato che deve essere mostrato sull'interfaccia; svolge pertanto il ruolo di contenitore di informazioni, ma implementa anche le logiche per dialogare con lo strato di business dell'applicazione stessa;

- Il **Controller** ha il compito di interpretare la richiesta dell'utente, di instanziare e valorizzare il model opportuno e, successivamente, di instradare la richiesta verso la view;

- La **View** è invece il template secondo cui il model deve essere rappresentato. Questo componente, pertanto, non contiene alcuna logica applicativa, ma solo l'eventuale logica necessaria per visualizzare correttamente il dato fornito come input.

In ASP.NET Core MVC, pertanto, il flusso di richiesta di una pagina e la generazione del markup di risposta non avviene in base alla composizione di controlli server side, come nel caso di ASP.NET Web Forms, ma tramite queste tre componenti del pattern Model-View-Controller. Per capire le modalità secondo cui questi oggetti

interagiscono, il modo migliore è quello di analizzare un progetto di esempio, che sarà argomento della prossima parte di questo capitolo.

I servizi di routing

Quando l'applicazione riceve una richiesta da parte di un browser, il runtime di ASP.NET Core, attraverso i suoi middleware, presentati nei capitoli precedenti, deve determinare un oggetto in grado di soddisfarla e di produrre il relativo markup HTML di risposta. Nel caso di ASP.NET Core MVC, questo oggetto prende il nome di **controller**. Come impareremo nelle prossime pagine, un controller possiede la caratteristica di esporre una serie di gestori per le differenti richieste che pervengono all'applicazione, denominate **action**.

Ovviamente, visto che le richieste all'applicazione avvengono sotto forma di un URL, è necessario specificare da qualche parte quali sono le corrispondenze tra essi e i controller/action necessari per soddisfarle. Per questa necessità ASP.NET MVC sfrutta l'infrastruttura di **routing**, che altro non è che un sistema attraverso il quale possiamo definire dei percorsi HTTP e come una chiamata a ciascuno di essi debba comportarsi.

Generalmente, questo avviene mediante l'utilizzo di un opportuno middleware, già introdotto nel Capitolo 3, e che qui riportiamo per riprendere il discorso nell'Esempio 4.1. Il contenuto di questo codice è all'interno del metodo `Configure` all'interno del file `Startup.cs`.

Esempio 4.1

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    …
    app.UseMvc(routes =>
    {
            routes.MapRoute(
                    name: "default",
                    template: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

Il mapping che per default viene inserito in ogni applicazione ASP.NET Core MVC, serve a gestire la mancanza della tipica corrispondenza tra un URL e un file fisico sul server, sostituita invece dall'individuazione del controller e della relativa action in base al valore delle due componenti sull'indirizzo.

> *La sintassi utilizzata per definire la regola di route sfrutta l'extension method* `MapRoute` *che, internamente, implementa 'interfaccia* `IRouteBuilder`, *alla base del routing di ASP.NET Core MVC.*

Grazie a questo middleware, quindi, una richiesta all'indirizzo */People/Index* verrà instradata presso un controller denominato `People`, e in particolare verso la sua action `Index`; analogamente, */Countries/Edit/1* corrisponderà alla action `Edit` del controller Countries, fornendo come parametro id il valore 1. Il metodo `MapRoute` ci permette anche di definire dei valori di default, nel caso qualcuno di questi parametri dovesse mancare. In virtù di questi parametri, alla root del sito `http://localhost/` corrisponderà il controller *Home* e la sua action *Index*. Ovviamente queste impostazioni possono essere personalizzate secondo la nostra necessità, sia modificando la route di default sia aggiungendo delle route ulteriori.

Ora che abbiamo compreso il meccanismo secondo il quale viene determinato il controller a cui demandare l'onere di soddisfare una richiesta, possiamo spostare la nostra attenzione su come effettivamente realizzarne uno.

Il controller e il model

Dal punto di vista strettamente pratico, un controller non è altro che una classe che implementa l'interfaccia `IController`. Sebbene sia quindi sufficiente creare una nuova classe all'interno del progetto, in Visual Studio possiamo anche usare la funzionalità **Add Controller**, presente nel menu contestuale di un progetto ASP.NET MVC, che apre la finestra di dialogo mostrata nella Figura 4.2.

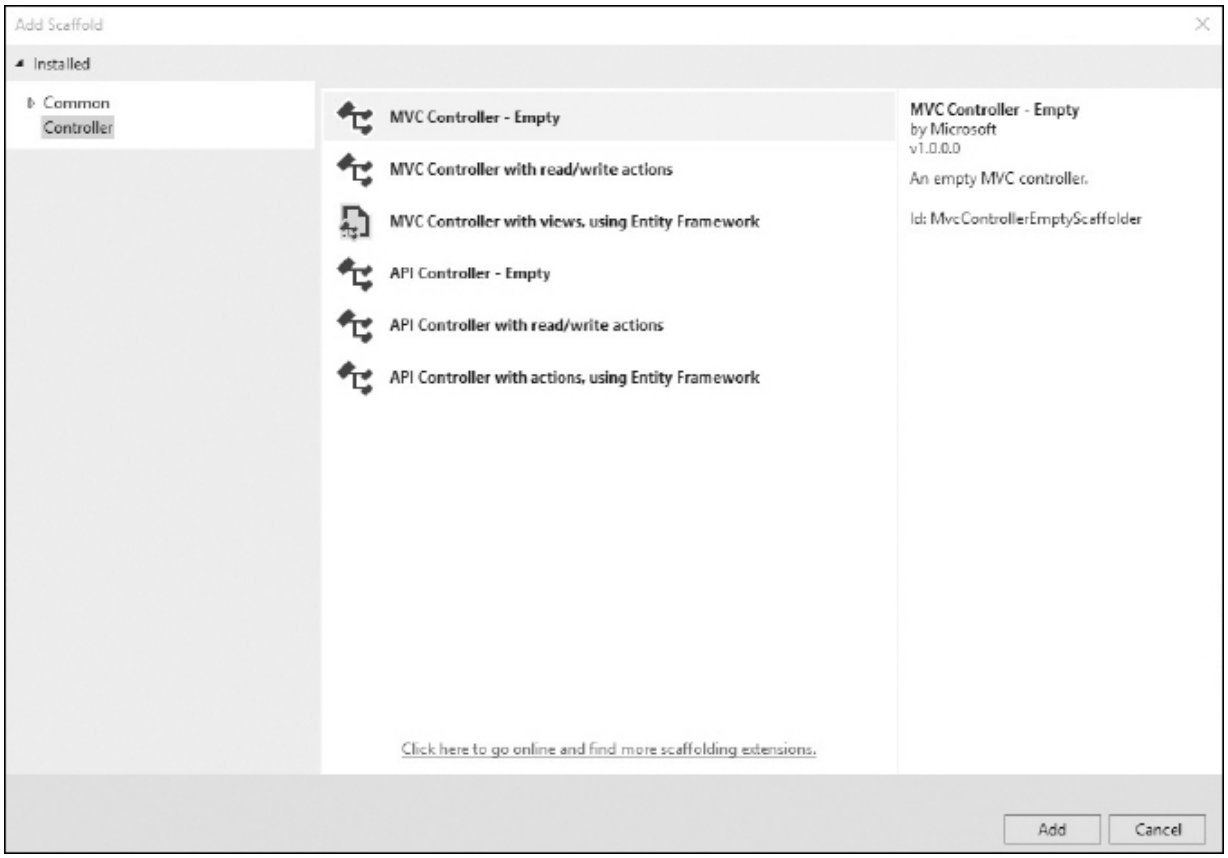


Figura 4.2 – Finestra di dialogo per la creazione di un nuovo controller.

Tramite questa maschera possiamo specificare il nome, il quale per convenzione viene fatto terminare con il suffisso *-Controller*, e il template che vogliamo utilizzare per generarlo. Se utilizziamo le impostazioni visibili nella figura, verrà aggiunta al progetto una nuova classe `HomeController`, il cui codice è quello dell'Esempio 4.2.

Esempio 4.2

```
public class HomeController : Controller
{
  public IActionResult Index()
  {
    return View();
  }
}
```

Come possiamo notare, `HomeController`, in realtà, eredita dalla classe base `Controller`, che internamente implementa già tutti i membri richiesti dall'interfaccia `IController`, e contiene la definizione della action `Index`, che non è altro che un metodo pubblico definito al suo interno; esso non presenta alcun tipo di logica e si limita a restituire come risultato una view e, in particolare, visto che non abbiamo specificato alcuna informazione addizionale, la **view predefinita** (individuata attraverso un meccanismo di convenzioni, che vedremo successivamente). Tipicamente, all'interno di una action abbiamo anche il compito di recuperare e valorizzare i dati che vogliamo rappresentare tramite la view. Come abbiamo accennato, il contenitore di questi dati è il **model**. Per casi semplici come quello che stiamo esaminando, la classe base `Controller` espone la proprietà `ViewBag`, ossia un oggetto di tipo `dynamic`, che è accessibile anche dalla view e che possiamo valorizzare con qualsiasi tipo di informazione, come nell'Esempio 4.3.

Esempio 4.3

```
public ActionResult Index()
{
  ViewBag.Message = ˝Ciao da ASP.NET Core˝;
  ViewBag.CurrentDate = DateTime.Now.ToString();
  ViewBag.ShowDate = true;

  return View();
}
```

All'interno del controller troverà spazio tutta la logica necessaria a recuperare i dati, manipolarli e poi rimandarli indietro al client, secondo la logica che ci siamo prefissati.

In realtà, in ASP.NET Core MVC non è obbligatorio, per un controller, né ereditare da una classe che implementi l'interfaccia `IController`, né restituire un tipo `IActionResult` come risultato di una action. In effetti, ASP.NET Core MVC supporta pienamente il concetto di controller POCO (Plain Old CLR Object - i vecchi cari oggetti del CLR). In estrema sintesi, un controller che non debba utilizzare i servizi di ASP.NET Core MVC può fare a meno di una classe base e può restituire qualsiasi tipo serializzabile.

Esempio 4.4

```
public class PocoController
{
  public Date CurrentTime()
  {
    return DateTime.Now;
  }
}
```

L'output della action dell'Esempio 4.4 sarà la serializzazione del tipo di ritorno – in questo caso una data – nel formato JSON. Diventa così molto più semplice, in estrema analisi, tenere action con scopi differenti all'interno dello stesso controller, semplificando la logica in quegli scenari in cui si fa ampio uso di endpoint che restituiscono JSON, per esempio in applicazioni SPA o AJAX. Avremo modo di tornare ampiamente su questi temi nel prossimo capitolo.

Giunti a questo punto, manca solo l'ultimo tassello per completare la nostra prima pagina: dobbiamo creare la nostra prima view.

La view e gli HTML helper

Come abbiamo più volte avuto modo di sottolineare durante il capitolo, la view è il componente del pattern MVC responsabile di rappresentare in forma di markup, visto che siamo nell'ambito di un'applicazione web, i dati contenuti all'interno del model.

In un progetto ASP.NET Core MVC, esse sono memorizzate all'interno della directory `Views`, disposta nella struttura di directory che vediamo nella Figura 4.3.

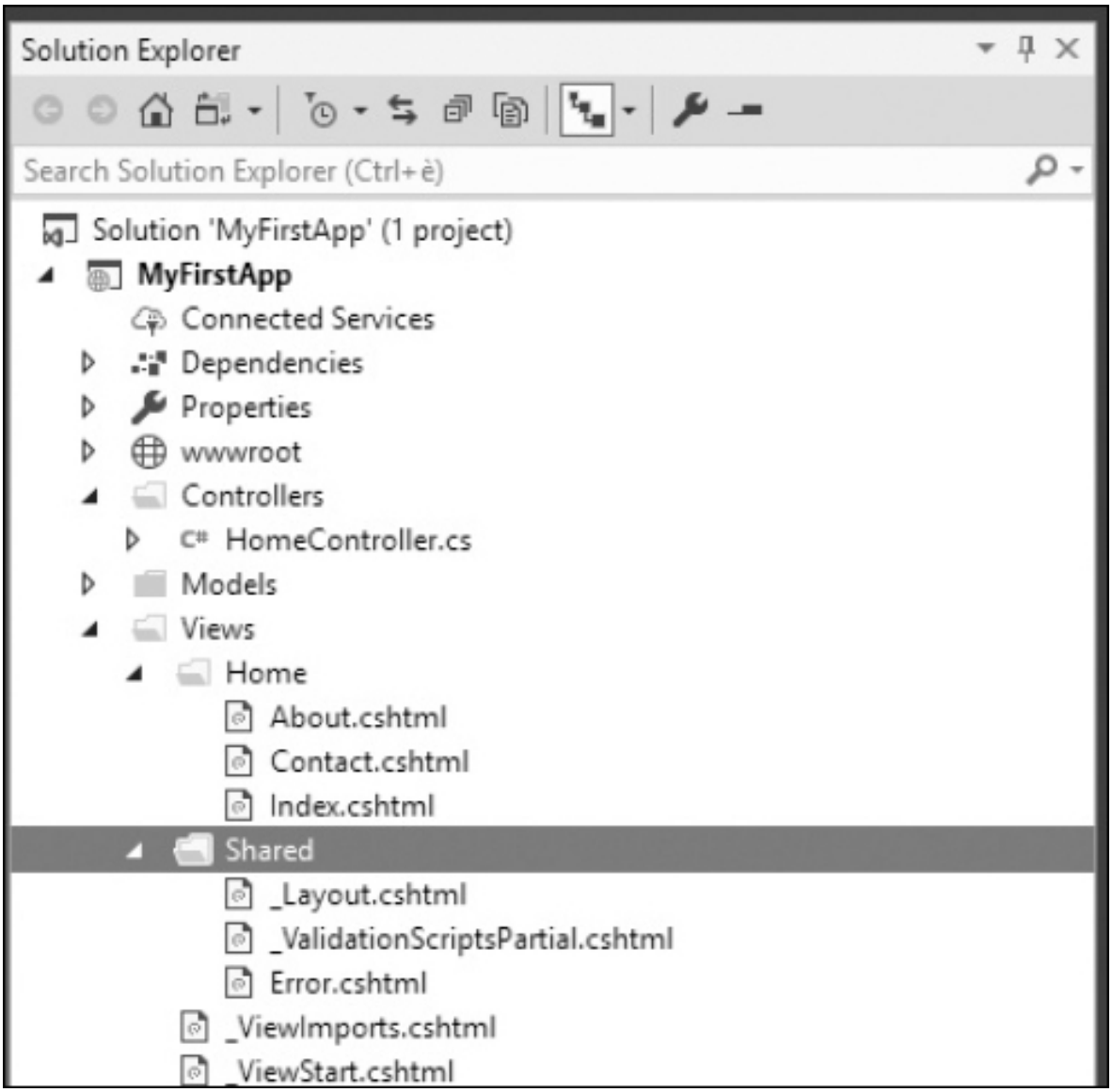


Figura 4.3 – Struttura di directory per le view.

La directory `Views`, a propria volta, contiene una sottodirectory per ogni controller. All'interno di queste ultime trovano posto i file, con estensione `.cshtml`: si tratta di file che vengono processati da un engine, denominato **Razor**, che, a seguito di un'operazione di parsing, produrrà delle classi (in C#) che rappresentano la struttura del nostro markup e che saranno poi eseguite effettivamente dal runtime.

Se utilizziamo Visual Studio, non dobbiamo preoccuparci di creare a mano la directory di un controller perché, anche in questo caso, Visual Studio ci mette a disposizione una comoda finestra di dialogo, mostrata

nella Figura 4.4, per generare una view. Per aprirla non dobbiamo far altro che selezionare l'opzione *Add View* dal menu contestuale che si ottiene dopo un click con il tasto destro sul codice della action.

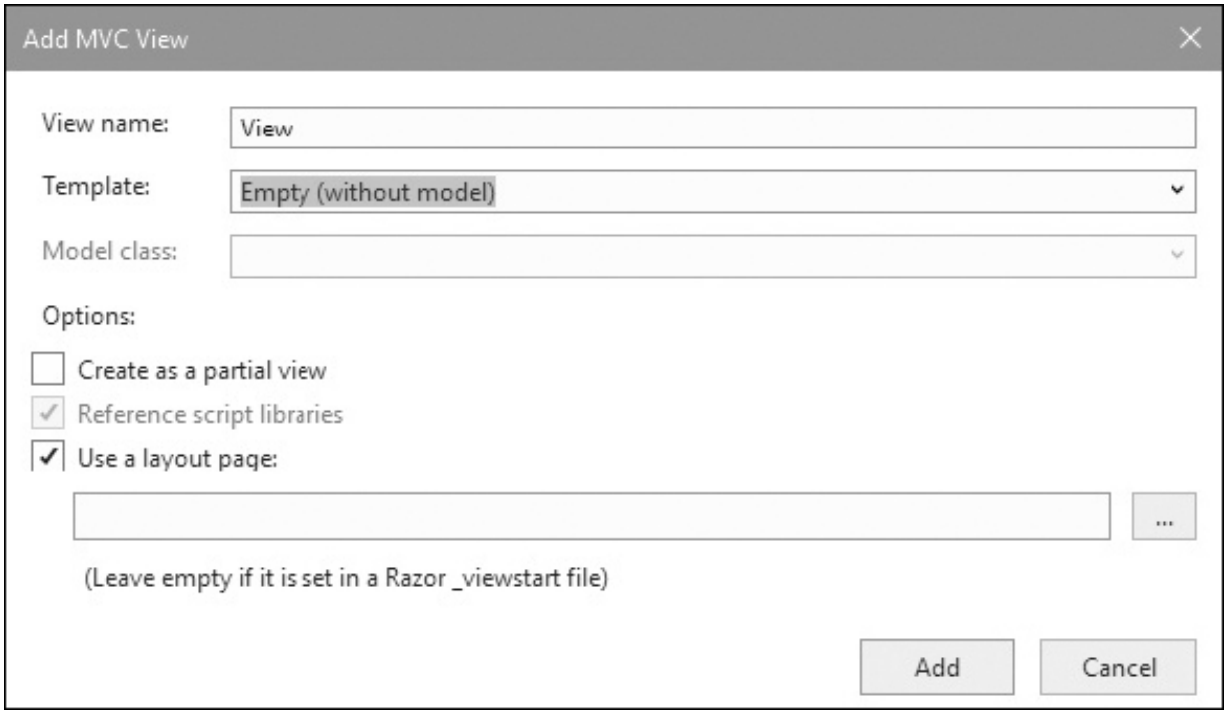


Figura 4.4 – Finestra di dialogo per la creazione di una nuova view.

Utilizzando un qualsiasi altro editor (come, per esempio, Visual Studio Code) l'operazione è fattibile manualmente, senza troppi patemi.

Come possiamo notare, il nome che ci viene proposto da Visual Studio coincide con quello della action da cui siamo partiti. In questo caso, prende il nome di **view predefinita** per quella determinata action, e potrà essere referenziata tramite il metodo `View`, come abbiamo visto nell'Esempio 4.4, senza specificare esplicitamente il nome. La dialog mostra anche ulteriori opzioni, di cui parleremo in maniera approfondita nel corso del Capitolo 6 e nei successivi. Per il momento, quindi, tralasciamole e proseguiamo effettuando un click sul pulsante *Add*; il risultato è la creazione di un nuovo file, il cui contenuto è mostrato nell'Esempio 4.5.

Esempio 4.5

```
@{
  ViewBag.Title = ″Index″;
}
<h2>Index</h2>
```

Come possiamo notare, si tratta di una sintassi davvero particolare, che concilia all'interno dello stesso file la presenza di markup HTML con il codice C#. A prescindere da quale linguaggio stiamo effettivamente utilizzando, nel codice precedente possiamo distinguere fondamentalmente due "zone":

- Un blocco di codice, delimitato da `@{ ... }` in C#, all'interno del quale accediamo al contenuto della variabile `ViewBag`, che abbiamo già avuto modo di conoscere nella sezione precedente, valorizzandone la proprietà `Title`.

- Una sezione di markup HTML, costituita da un tag `H2`.

Nel codice di una view, grazie al carattere `@`, possiamo cambiare il contesto da codice a markup, e viceversa.

Esempio 4.6

```
<h2>Index</h2>
<p>@ViewBag.Message</p>
@if (ViewBag.ShowDate)
{
  <div>La data corrente è @ViewBag.CurrentDate</div>
}
@Html.ActionLink(″Un link...″, ″SayHello″)
```

Il codice dell'Esempio 4.6 mostra alcune peculiarità della sintassi di Razor.

- Come abbiamo accennato, una volta inserito il tag `<p>` (contesto markup), possiamo passare al contesto codice tramite il carattere `@`

e referenziare il contenuto di `ViewBag`, che abbiamo valorizzato nel controller;

- nel realizzare il template, potremmo aver bisogno dei tipici costrutti di branching o di iterazione e, per usufruirne, non dobbiamo far altro che passare al contesto del codice. Per esempio nel codice precedente abbiamo utilizzato un blocco `if` per visualizzare o meno la data corrente al variare del parametro `ShowDate`;

- l'istruzione `@Html.ActionLink` appartiene a una categoria di metodi parecchio utilizzati nella realizzazione di pagine con Razor, e che sono denominati **HTML helper**. Nello specifico, `ActionLink` serve a generare un link verso la action "SayHello" dello stesso controller, con il testo specificato; sebbene possiamo comunque inserire un link sfruttando il tag `<a>`, questo metodo è preferibile perché l'URL generato dipende dalle impostazioni del routing e, nel caso queste vengano cambiate in futuro, anche il link verrà modificato automaticamente di conseguenza;

- l'engine è sufficientemente evoluto da riuscire a discernere i casi in cui l'utilizzo del carattere `@` non implica un cambio di contesto.

Il risultato di questa nostra prima pagina realizzata con ASP.NET Core MVC è visibile nella Figura 4.5.

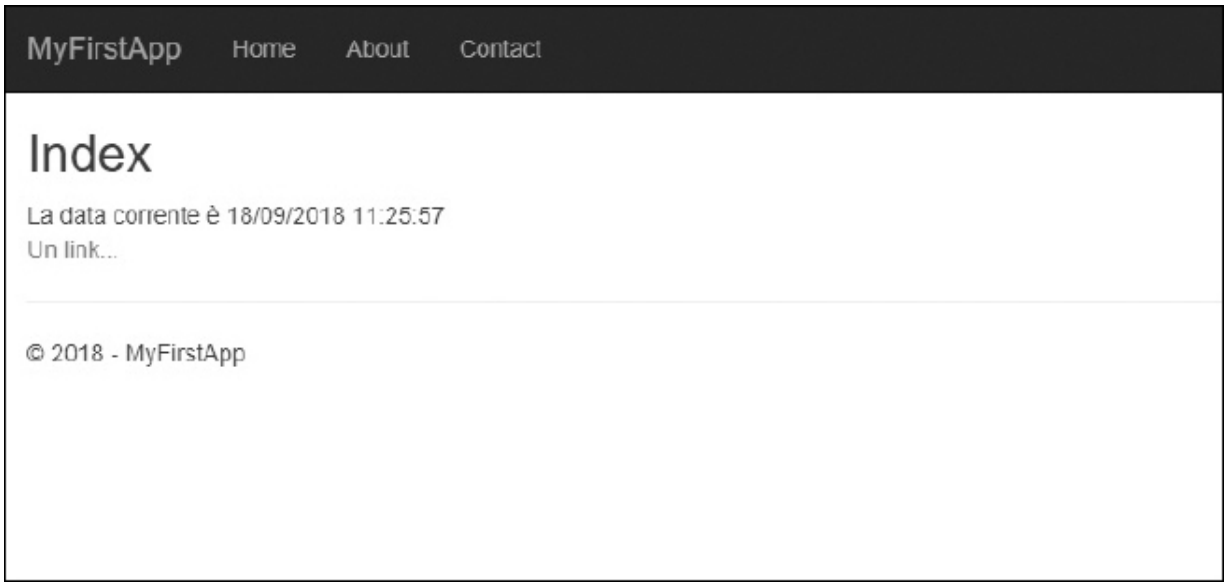


Figura 4.5 – La nostra prima pagina in ASP.NET Core MVC.

Se proviamo a dare un'occhiata al sorgente della pagina, mostrato nell'Esempio 4.6, possiamo effettivamente toccare con mano la sostanziale differenza rispetto ad altri framework più complessi: il contenuto della pagina è molto semplice, non sono presenti view state o control state, e il markup generato è assolutamente sovrapponibile a quanto abbiamo scritto nella view.

Esempio 4.7 - HTML

```
<!DOCTYPE html>
<html>
  <head>
    <!-- altro markup qui -->
  </head>
  <body>
    <h2>Index</h2>
    <p>Ciao da ASP.NET MVC</p>
    <div>La data corrente è 15/06/2018 11:48:11</div>
    <a href="/Home/SayHello">Un link...</a>
    <script src="/Scripts/jquery.js"></script>
  </body>
</html>
```

Come possiamo notare, esistono comunque degli elementi addizionali, estranei alla view che abbiamo realizzato, come la sezione `<head>` della

pagina o il riferimento alla libreria jQuery. Essi sono stati inseriti da quella che in ASP.NET Core MVC è chiamata **layout page** e che si trova all'interno della directory **`Views\Shared`**, è chiamata `_Layout.cshtml` e il cui contenuto tipo è quello dell'Esempio 4.8.

Esempio 4.8

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width" />
  <title>@ViewBag.Title</title>
  @Styles.Render("~/Content/css") </head>
<body>
  @RenderBody()
  @Scripts.Render("~/bundles/jquery")
  @RenderSection("scripts", required: false)
</body>
</html>
```

In questa fase non è necessario entrare nel dettaglio di ogni singola riga di codice di questa view, di cui parleremo approfonditamente nel corso dei prossimi capitoli. L'unico aspetto da notare, al momento, è la presenza del metodo `RenderBody`, in corrispondenza del quale verrà inserito il markup prodotto dalla specifica view.

Nonostante si sia trattato di un esempio assolutamente banale, ci è comunque servito ad apprendere molto del funzionamento e delle logiche che regolano il modello di ASP.NET Core MVC: abbiamo visto come, alla ricezione di una richiesta, il primo passaggio eseguito dal runtime sia l'instradamento tramite routing verso un controller e, più in dettaglio verso una action; quest'ultima, come risultato, ha restituito una view, ossia un template che abbiamo costruito integrando markup e codice, tramite la particolare sintassi dell'engine Razor.

Al momento, però, non siamo ancora in grado di gestire l'input dell'utente, per esempio un form che possa effettuare un POST verso il nostro sito web. Nella prossima sezione proveremo a colmare questa lacuna.

Gestire una form di input

Il funzionamento di ASP.NET Core MVC, di fronte all'input che arriva da una form, è ancora una volta orientato alla semplicità: è privo di astrazioni e maggiormente aderente al funzionamento del protocollo HTTP in generale. Una richiesta che proviene da una form viene gestita in maniera analoga a tutte le altre, individuando la coppia controller/action corrispondente e processandola.

Cerchiamo di chiarire meglio il concetto con un esempio. Per poter consentire all'utente di inserire dei dati, intanto dobbiamo predisporre una pagina apposita, e quindi una action come la seguente.

Esempio 4.9

```
public ActionResult SayHello()
{
  return this.View();
}
```

Il codice dell'Esempio 4.9 è assolutamente banale e non merita alcun commento; ben più interessante è invece il contenuto della view, mostrato nell'Esempio 4.10.

Esempio 4.10

```
@using (Html.BeginForm())
{
  <div>
    <p>Inserisci il tuo nome: </p>
    <p>@Html.TextBox("name")</p>
    <input type="submit" value="Go!" />
  </div>
}
<div>@ViewBag.Message</div>
```

Nel codice in alto abbiamo utilizzato un paio di HTML helper che, nello sviluppo di applicazioni ASP.NET Core MVC, si rivelano sicuramente utili. Il primo di questi helper è `BeginForm`, che serve a produrre il tag `<form>`; la modalità di utilizzo è leggermente diversa a quella di `ActionLink`, che

abbiamo visto in precedenza, visto che va inserito all'interno di un blocco `using`, così che abbiamo la possibilità di specificare con esattezza anche il punto in cui termina. Successivamente, abbiamo sfruttato l'HTML helper `TextBox`, per generare un tag `<input type="text" />` all'interno del quale l'utente potrà inserire il suo nome.

Al click sul bottone di submit, il browser dell'utente invierà in POST il contenuto della form al medesimo indirizzo della richiesta precedente, ossia */Home/SayHello*. Dal nostro punto di vista, questo si tradurrà nell'esecuzione di un'ulteriore action che, in base alle regole di routing, dovrà comunque chiamarsi `SayHello`, ma sarà specifica per gestire la chiamata in POST.

Esempio 4.11

```
[HttpPost]
public ActionResult SayHello(string name)
{
  ViewBag.Message = string.Format("Ciao {0}!", name);
  return this.View();
}
```

Anche in questo caso, ci sono alcuni aspetti da sottolineare: questa action, seppure omonima a quella che abbiamo realizzato in precedenza, è marcata con l'attributo `HttpPost`, in modo da segnalare che dovrà essere utilizzata per gestire le richieste di tipo POST. Inoltre, come possiamo notare, essa presenta un parametro `name`, che poi abbiamo utilizzato per produrre il messaggio di risposta. Come vedremo nel corso del Capitolo 15, è compito del runtime di ASP.NET Core MVC, e in particolare di un oggetto denominato model binder, quello di valorizzare questi parametri in base al contenuto della richiesta. Per ora, ci basta sapere che, in virtù del fatto che il suo nome corrisponde a quello che abbiamo utilizzato nell'html helper `TextBox`, esso conterrà effettivamente il dato inserito dall'utente nella form.

Dopo aver valorizzato opportunamente la `ViewBag`, l'ultima riga di codice dell'Esempio 4.11 restituisce la stessa view `SayHello.cshtml` dell'Esempio 4.10, così che il messaggio possa effettivamente essere mostrato.

Ancora una volta, si è trattato di un esempio piuttosto semplice (il classico "hello world"), ma che ci aiuta a capire i meccanismi basilari di ASP.NET Core MVC anche nel caso della gestione dell'input utente. Sotto questo punto di vista, ovviamente, c'è ancora tanto da dire, a partire dalle modalità per realizzare form complesse fino ad arrivare a spiegare come impostare regole di validazione. Ci occuperemo in maniera approfondita di questo argomento nel corso del Capitolo 15.

Per il momento, invece, cambiamo momentaneamente argomento e torniamo a parlare della struttura di un progetto ASP.NET Core MVC e di come possiamo organizzare le varie componenti di un progetto, nel momento in cui la complessità dell'applicazione e il numero di pagine aumentano.

ASP.NET Core MVC e progetti complessi: le aree

Come abbiamo avuto modo di vedere nel corso di questo capitolo, in ASP.NET Core MVC la struttura di directory del progetto non rispecchia l'effettivo path delle pagine, che invece è determinato esclusivamente dalle regole di routing.

Al contrario, esse hanno esclusivamente la funzione di contenitori, secondo una struttura ben definita che stabilisce dove posizionare i vari file di model, controller e view.

> *A rigor del vero, bisogna comunque specificare che il framework impone che le convenzioni sulle directory siano rispettate solo per quanto riguarda le view, mentre non è necessario che model e controller si trovino nelle directory omonime. I concetti espressi da questa sezione rimangono comunque validi.*

Questa impostazione, però, può costituire un problema quando le dimensioni del progetto aumentano, a causa del proliferare di file, o anche quando si vuole suddividere le varie funzionalità in "aree tematiche". Pensiamo, per esempio, al caso di un CMS, in cui magari abbiamo sia una parte pubblica, visibile a tutti, sia una sezione di backoffice amministrativo, anche con differenti layout, pattern di routing

e regole di accesso: si tratta delle tipiche necessità per le quali, in ASP.NET MVC, è stato introdotto il concetto di **Area**, che troviamo anche all'interno di ASP.NET Core MVC. Da Visual Studio, se effettuiamo un click con il tasto destro del mouse sul progetto, dal menu contestuale possiamo aggiungere una nuova area tramite il comando *Add Area* che vediamo nella Figura 4.6.

A questo punto, se diamo un nome alla nuova area, per esempio Backoffice, e confermiamo, viene creata nel progetto una nuova struttura di directory, del tutto simile alla principale, come mostrato nella Figura 4.7.

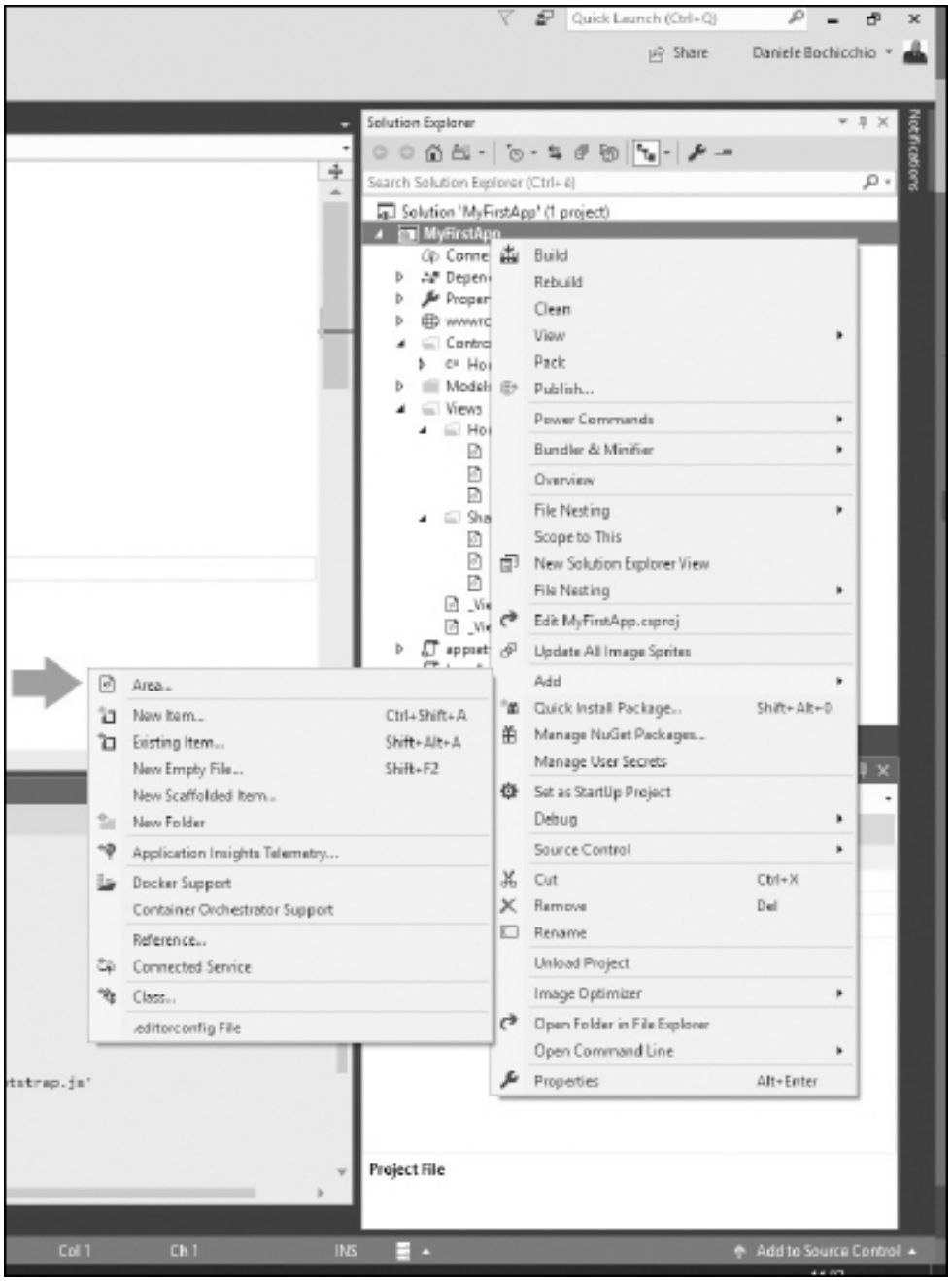


Figura 4.6 – Menu contestuale per l'aggiunta di un'area.

Non utilizzando Visual Studio, l'operazione si può portare a termine facilmente, creando manualmente una struttura come quella riportata nell'immagine.

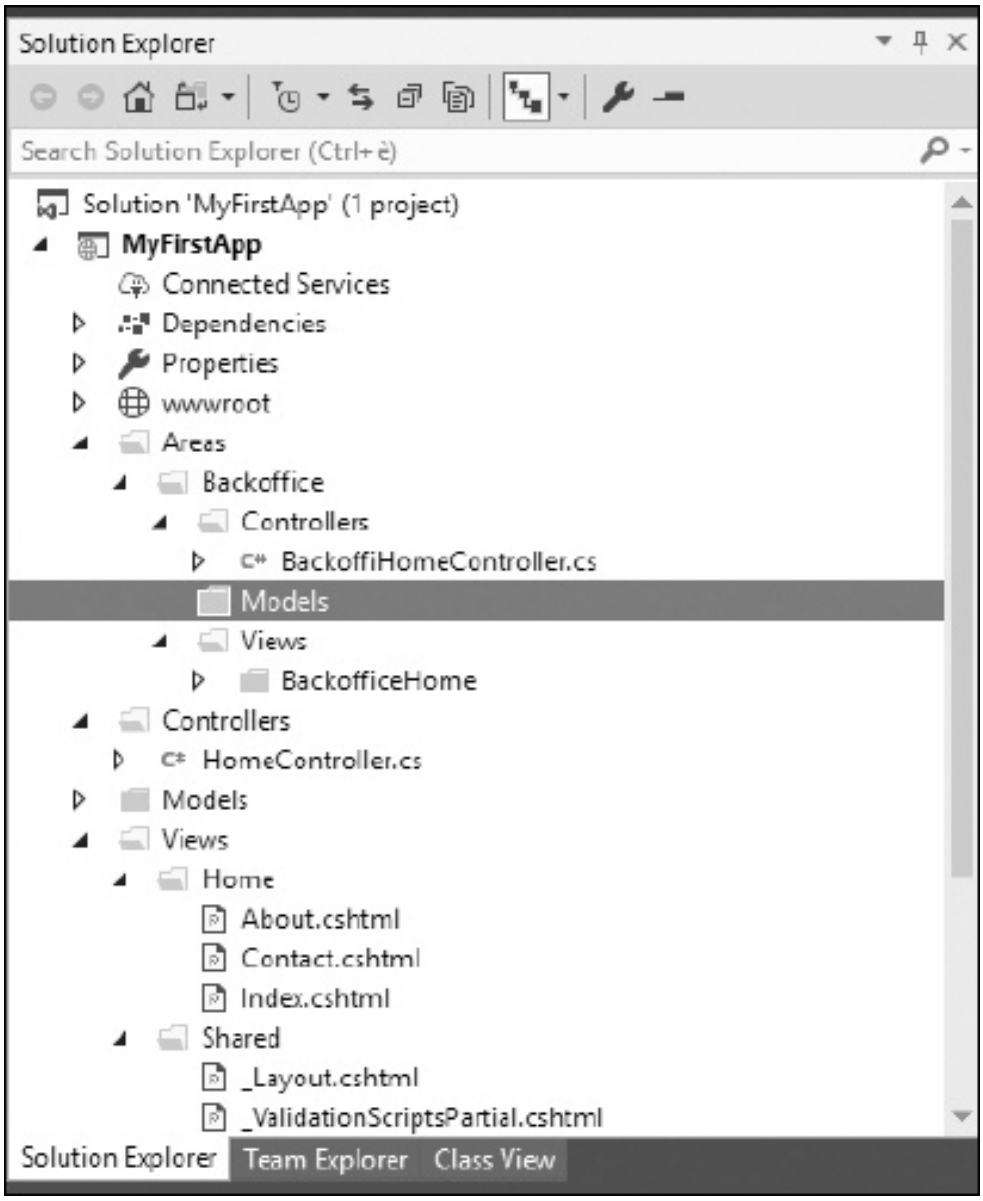


Figura 4.7 – La struttura delle directory dell'area Backoffice.

In questo modo abbiamo definito una sezione, separata dal resto del progetto, all'interno della quale implementare la nuova area funzionale dell'applicazione: abbiamo una directory specifica per controller e model, oltre alla struttura relativa alle view. Questo ci consente, per esempio, di definire una layout page specifica, che sarà utilizzata solo dalle pagine dell'area backoffice.

Per completare l'operazione e fare in modo che tutto funzioni, dobbiamo leggermente cambiare la parte relativa al routing, per

supportare una nuova convenzione, basata sull'aggiunta di un'area.

Esempio 4.12

```
app.UseMvc(routes =>
{
  route.MapRoute(
    name : ″areas″,
    template : ″{area:exists}/{controller=Home}/{action=Index}/{id?}″
  );
  …
  // mantenere il routing già presente
});
```

La registrazione del routing per le aree va fatta una sola volta e deve precedere quella più generale, che abbiamo visto prima, perché le stesse agiscono secondo un algoritmo di corto circuito: la prima vera fa saltare la valutazione delle altre regole, quindi quella più generica va tenuta sempre per ultima. La ricerca di una view verrà effettuata secondo questa logica:

- `/Areas/<Area-Name>/Views/<Controller-Name>/<Action-Name>.cshtml`
- `/Areas/<Area-Name>/Views/Shared/<Action-Name>.cshtml`
- `/Views/Shared/<Action-Name>.cshtml`

Inoltre, a differenza di ASP.NET MVC, in ASP.NET Core MVC è necessario decorare ciascun controller con l'attributo `Area`, come nell'esempio che segue.

Esempio 4.13

```
[Area(″Backoffice″)]
public class AdminController : Controller
{
  public IActionResult Index()
  {
    return View();
  }
}
```

Nel corso di questo capitolo abbiamo accennato all'HTML helper `ActionLink`, tramite cui possiamo generare dei link verso altre pagine dell'applicazione, basandoci sul nome del controller e della action piuttosto che sull'indirizzo fisico. In generale, questo metodo **punta sempre all'area della pagina corrente**. Non esiste uno specifico overload che ci consenta di specificare il nome dell'area, pertanto se vogliamo creare un link dall'area principale all'area di backoffice, dobbiamo utilizzare la tecnica dell'Esempio 4.14.

Esempio 4.14

```
@Html.ActionLink("Vai al backoffice", "Index", "Admin",
  new { Area = "Backoffice" }, null)
```

Nel codice precedente abbiamo sfruttato un **anonymous type** per specificare un parametro addizionale del routing, ossia il nome dell'area. Il risultato in pagina, pertanto, sarà un link alla action `Index` di `AdminController`, contenuto nell'area denominata Backoffice.

Conclusioni

In questo capitolo abbiamo voluto introdurre i concetti basilari di ASP.NET Core MVC, che poi verranno esplorati in maggiore dettaglio nel prosieguo del libro. ASP.NET Core MVC è il framework per lo sviluppo di applicazioni web, basato sul popolare pattern Model-View-Controller, che impareremo a utilizzare con ASP.NET Core. Nei fatti, quando parliamo di ASP.NET Core, al 90% ci riferiamo anche ad ASP.NET Core MVC, che rappresenta il servizio certamente più utilizzato fra quelli messi a disposizione.

Dopo aver visto le componenti che lo contraddistinguono e le responsabilità di ognuna, abbiamo creato il nostro primo progetto in Visual Studio, sfruttando uno dei molteplici template presenti, per poi dedicarci alla creazione della nostra prima pagina. Si è trattato di un esempio molto semplice, che però ci ha fatto apprezzare la semplicità

del modello di sviluppo basato sul pattern MVC e il totale controllo che abbiamo sul markup effettivamente prodotto in pagina. Anche per quanto riguarda la gestione delle form di input, sebbene abbiamo solo iniziato a illustrare la problematica, che sarà affrontata in maniera approfondita nel capitolo successivo, possiamo già renderci conto di come la filosofia di base non cambi, dovendo comunque esporre una action a cui è demandata la gestione della request di POST.

Come ultimo argomento del capitolo abbiamo introdotto il concetto di area, tramite la quale possiamo partizionare e organizzare i file di progetto in diverse aree funzionali, ognuna caratterizzata dai propri controller, view e regole di routing.

A questo punto abbiamo tutte le nozioni di base necessarie per affrontare maggiormente nel dettaglio le varie caratteristiche di ASP.NET Core MVC, a partire dal prossimo capitolo, in cui ci occuperemo dei controller.

5

Controller e routing

URL Routing in ASP.NET Core MVC

Configure Startup
RouteMiddleware

Esempio 5.1

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
  if (env.IsDevelopment())
  {
    app.UseDeveloperExceptionPage();
    app.UseBrowserLink();
  }
  else
  {
    app.UseExceptionHandler("/Home/Error");
  }
  app.UseStaticFiles();
  app.UseMvc(routes =>
  {
    routes.MapRoute(
      name: "default",
      template: "{controller=Home}/{action=Index}/{id?}");
  });
}
```

MapRoute

IRouteBuilder

RouteMiddleware

RouteAsync

RouteContext.Handler

Next

Constraint sulle route

MapRoute

IRouteConstraint

MapRoute

Esempio 5.2

```
routes.MapRoute(
  name: ″numeric-id-route″,
  template: ″{controller=Home}/{action=Index}/{id?}″,
  constraints: new { id = ″\\d+″ },
  defaults: null
);
```

Esempio 5.3

```
routes.MapRoute(
  name: ″numeric-id-route″,
  template: ″{controller=Home}/{action=Index}/{id:int}″);
```

{id:int}

- int
- bool true false
- datetime
- decimal
  decimal

- double
  double
- float float
- long long
- guid
- minlength(value) maxlength(value)
- length(value) length(min, max)
- min(value) max(value)
- range(min, max)
- alpha
- regex(expression)
- required

IRouteConstraint

Esempio 5.4

```
routes.MapRoute(
  name: ″users-profiles″,
  template: ″users/{username:length(3,15)}″,
  defaults: new {controller=″Users″, action=″Profile″});
```

Users
username
Profile Users
username

Anatomia di un controller

IController
Execute

Esempio 5.5

```
public interface IController
{
  void Execute(RequestContext requestContext);
}
```

Execute

Request
Response HttpContext

Controller
IController

Controller

Proprietà e metodi di supporto della classe Controller

HttpContext
Controller

Tabella 5.1 – Contesto di richiesta nella classe Controller.

Nome proprietà	Descrizione
HttpContext	HttpContext
ControllerContext	HttpContext
Request	
Response	
User	

ViewData ViewBag

ViewBag dynamic

Esempio 5.6

```
public IActionResult Index()
{
  // ViewData e ViewBag agiscono sul medesimo dizionario
  ViewBag.Message = ″Esempio di messaggio″;
  ViewData[″Message″] = ″Sostituisce il messaggio precedente″;
  return View();
}
```

object

Ciclo di vita di un controller

Dispose

DbContext

Esempio 5.7

```
public class HomeController : Controller
{
  private NorthwindContext _context;
  public HomeController()
  {
    _context = new NorthwindContext();
```

```
    }
    protected override void Dispose(bool disposing)
    {
      if (_context != null)
        _context.Dispose();
      base.Dispose(disposing);
    }
  }
```

In questo esempio stiamo utilizzando una particolare funzionalità, chiamata controller activator, che è responsabile di istanziare i controller quando necessario. Questo è in grado di gestire dipendenze e, pertanto, di valorizzare eventuali parametri del costruttore. L'esempio qui mostrato è in realtà poco corretto da un punto di vista pratico e vedremo come sia possibile sostituirlo con uno che faccia uso di IoC container più tardi in questo stesso capitolo.

Tabella 5.2 – Metodi personalizzabili della classe

`Controller`

Nome metodo	Descrizione
`OnActionExecutionAsync`	`async`
`OnActionExecuting`	
`OnActionExecuted`	

Uso della Dependency Injection nei controller

"High level modules should not depend on low level modules; both should depend on abstractions."
- Dependency Inversion Principle

"Moduli di alto livello non dovrebbero dipendere da moduli di basso livello; entrambi dovrebbero dipendere da astrazioni." - Principio di Dependency Inversion

Esempio 5.8

```
public interface IMyService
{
  IEnumerable<Person> GetPeople();
}
public class MyFakeService : IMyService
{
  public IEnumerable<Person> GetPeople()
  {
    for (int i = 0; i < 10; i++)
    {
      yield return new Person
      {
        FirstName = ″Daniele″,
        LastName = ″Bochicchio ″ + i,
      };
    }
  }
}
```

MyFakeService

Startup

ConfigureServices

Esempio 5.9

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddMvc();
  // i nostri servizi applicativi dopo MVC
  services.AddScoped<IMyService, MyFakeService>(); // interfaccia -> classe
  services.AddScoped<PeopleContext>();             //solo lifetime
}
```

PeopleContext

- AddTransient
- AddScoped
- AddSingleton

Esempio 5.10

```
public class HomeController : Controller
{
  private readonly IMyService myService;
  public HomeController(IMyService myService)
  {
    this.myService = myService;
  }
  public IActionResult Index()
  {
    var model = myService.GetPeople();
    return View(model);
  }
  public IActionResult Search([FromServices]IMyService svc)
  {
    var model = svc.GetPeople();
    return View(model);
  }
}
```

myService

MyFakeService

Search

FromServices

RequestServices

HttpContext

MyFakeService

IMyService

ConfigureServices

Startup

Utilizzare un motore di IoC container differente

Autofac Autofac.Extensions.DependencyInjection ConfigureServices Startup

Esempio 5.11

```
public IServiceProvider ConfigureServices(IServiceCollection services)
{
  services.AddMvc();
  // autofac
  var containerBuilder = new ContainerBuilder();
  containerBuilder.RegisterModule<DefaultModule>();
  containerBuilder.Populate(services);
  var container = containerBuilder.Build();
  return new AutofacServiceProvider(container);
}
```

DefaultModule

Module

Controller

action

La action come gestore della richiesta

IController

Controller

action

http://localhost/Home/Index

Esempio 5.12

```
public class HomeController : Controller
{
  public IActionResult Index()
  {
    return View();
  }
  public IActionResult ActionWithParams(int id, string search,
    int value = 0)
  {
    // con il routing standard:
    // {controller}/{action}/{id}
    // id -> route data
    // search e value -> query string o form
    return View();
  }
}
```

ActionWithParams

http://localhost/ActionWithParams/5?search=test

id

5
search test

value null
int
nullable value type int?

IActionResult

L'interfaccia IActionResult e i diversi tipi di risposta

IActionResult
IActionResult
ExecuteResultAsync

ActionResult
ExecuteResult

Controller

View

ViewResult

In generale, è possibile costruire una action che restituisca anche un semplice oggetto, come una stringa o un numero intero. In questo caso sarà il framework stesso a creare un risultato wrapper, di tipo ContentResult o JsonResult, che conterrà come risposta la rappresentazione tramite il metodo ToString dell'oggetto stesso (se semplice) o la sua serializzazione in formato JSON. Questo è il meccanismo utilizzato anche dai controller POCO per generare l'output, ma consente di avere nello stesso controller anche action che restituiscano il risultato in formato JSON, così da essere più facilmente utilizzabili come endpoint in applicazioni SPA/AJAX.

espandibilità

ViewResult

I tipi ViewResult e PartialViewResult

ViewResult

View

view predefinita

Views\[NomeController] Views\Shared

Views\Home\Index.cshtml

Anche questo è un comportamento del tutto personalizzabile e dettato dall'implementazione standard del view engine di ASP.NET Core MVC.

View

Esempio 5.13

```
public IActionResult About()
{
  if (DateTime.Today.DayOfWeek == DayOfWeek.Sunday)
    return View("SundayAbout");
  return View();
}
```

model

ViewBag ViewData

Person

Esempio 5.14

```
public IActionResult Search(int id)
{
  var model = new Person()
  {
    FirstName = "Daniele",
```

```
      LastName = ″Bochicchio″
    };
    return View(model);
  }
```

ViewResult View
PartialViewResult PartialView

layout page

Esempio 5.15

```
  public IActionResult Contact(bool embed = false)
  {
    if (embed)
      return PartialView();
    else
      return View();
  }
```

I tipi RedirectResult, RedirectToRouteResult e HttpStatusCodeResult

RedirectResult

Esempio 5.16

```
public IActionResult GoToGoogle()
{
  return Redirect("http://www.aspitalia.com");
}
```

Redirect

302 (temporary redirect)

RedirectPermanent

301 (permanent redirect)

RedirectToAction

Esempio 5.17

```
public IActionResult BackToIndex()
{
  return RedirectToAction(nameof(Index));
}
public IActionResult ComplexRedirect()
{
  return RedirectToAction(
    nameof (Details),
    nameof(CustomersController).Replace("Controller", ""),
    new { id = 5 });
}
```

RedirectToRouteResult Index

http://localhost/Customers/Details/5

RedirectToActionPermanent

nameof

L'uso di nameof con i controller necessita che la stringa venga ripulita del suffisso controller, presente per convenzione. In un'applicazione reale questo replace in linea può essere sostituito dall'uso di un extension method. Un altro approccio è quello di non specificare il suffisso Controller e dare un nome più semplice ai controller, aggiungendo l'attributo [Controller] alla classe.

Person

Esempio 5.18

```
public IActionResult Find(int id)
{
  Person person = null; // aggiungere la ricerca su database...
  if (person == null)
    return HttpNotFound();
  return View(person);
}
```

HttpStatusCodeResult
HttpNotFound

Restituire JSON con JsonResult

JsonResult

Esempio 5.19

```
public IActionResult GetPerson(string firstName, string lastName)
{
  var result = new Person
  {
    FirstName = firstName,
    LastName = lastName
  };
  return Json(result, JsonRequestBehavior.AllowGet);
}
```

GET POST PUT DELETE

JsonRequestBehavior.AllowGet

Esempio 5.20

```
public Person GetPerson(string firstName, string lastName)
{
  return new Person
  {
    FirstName = firstName,
    LastName = lastName
  };
}
```

Il tipo FileResult e il metodo File

File controller

Esempio 5.21

```
public IActionResult Download()
{
  return File(″file.xlsx″, ″application/excel″, ″export.xlsx″);
}
```

byte

Il tipo ContentResult e il metodo Content

ActionResult

ContentResult

Content

Esempio 5.22

```
public IActionResult GetSomeText()
{
  string result = ″Lorem ipsum...″;
  return Content(result, ″text/plain″, Encoding.UTF8);
}
```

Controllo dell'esecuzione di una action

Esempio 5.23

```
[HttpGet]
public IActionResult GetOnly()
{
  return View();
}
[HttpPost]
public IActionResult PostOnly()
{
  return View();
}
[ActionName(″ActualName″)]
public IActionResult ThisIsNotTheName()
{
  return View();
}
[NonAction]
public object NotAnAction()
{
  return null;
}
```

HttpGetAttribute HttpPostAttribute

ThisIsNotTheName

ActionNameAttribute

NonActionAttribute

Esecuzione asincrona di una action

async/await

HttpContext

async

Esempio 5.24

```
public async Task<IActionResult> AsyncAction()
{
  WebClient client = new WebClient();
  string s = await
    client.DownloadStringTaskAsync(″http://www.servizio.remoto″);
  return Content(s);
}
```

Task

Task<IActionResult>

await

DownloadStringTaskAsync

Task Task<T>

Il pattern async/await è alla base delle ultime versioni di C#. Benché fuori dagli scopi di questo libro, consigliamo di approfondire l'argomento attraverso la lettura di questo articolo su: http://aspit.co/ai9.

L'infrastruttura dei filtri

filtri

- IActionFilter
- IResponseFilter
- IExceptionFilter
- IAuthorizationFilter

La classe Controller implementa solo l'interfaccia IActionFilter (oltre che IAsyncActionFilter) e, quindi, è a tutti gli effetti un filtro; questa classe ci consente di gestire le varie casistiche tramite una serie di metodi virtuali, che abbiamo visto nelle prime pagine di questo capitolo. Rispetto ad ASP.NET MVC, in ASP.NET Core MVC la classe controller non implementa le altre interfacce, perché l'infrastruttura dei middleware è abbastanza sovrapponibile ai filtri e questi servizi, più generici, non sono implementati con questo approccio.

HTTPS

RequireHttpsAttribute

Esempio 5.25

```
public class HomeController : Controller
{
  [RequireHttps]
  public IActionResult SecureAction()
  {
    return Content(″Secure content″);
  }
}
[RequireHttps]
public class SecureController : Controller
{
  public IActionResult Index()
  {
    return Content(″Secure content″);
  }
}
```

Conclusioni

Controller

IActionResult

6

Le view e Razor

Nello scorso capitolo abbiamo approfondito il ruolo che i controller e le action hanno nell'ambito di un'applicazione ASP.NET Core MVC nell'interpretare le varie richieste che arrivano al server e generare il tipo di risposta più opportuno. Al di là delle numerose possibilità che abbiamo, il risultato più comune per una action dovrà essere markup HTML e il componente incaricato di generare markup in ASP.NET Core MVC è una view.

In questo capitolo introdurremo innanzi tutto Razor, ossia l'engine tramite il quale potremo scrivere il codice delle view, e vedremo come la sua sintassi ci permetta di realizzare dei template in maniera parecchio versatile e leggibile. Successivamente ci occuperemo di tutti quegli strumenti che ci vengono forniti dal framework per rendere il più agevole possibile il nostro lavoro: in particolare parleremo delle layout view, tramite i quali possiamo dare un look uniforme alle pagine del nostro sito e supportare diverse tipologie di device, realizzando interfacce ad-hoc per ognuno di essi.

Poi introdurremo il concetto di HTML e tag helper, lo strumento proposto da ASP.NET Core MVC che ci consente di scrivere il markup delle pagine lavorando a un livello più alto di astrazione: in particolare, introdurremo il funzionamento di alcuni di essi (molti helper riguardano in maniera più specifica la realizzazione di form di input e, pertanto, li ritroveremo nel capitolo successivo), con un particolare occhio anche a partial view e child action, grazie alle quali possiamo creare componenti dell'interfaccia riutilizzabili in molte pagine.

Creare view in ASP.NET Core MVC grazie a Razor

Nel capitolo precedente abbiamo accennato al fatto che una view è quel componente del pattern MVC che ha il compito, nel caso di un'applicazione web, di rappresentare il model tramite markup HTML, in modo che poi possa essere servito al browser dell'utente. Dal punto di vista strettamente pratico, però, una view è un file posizionato all'interno di una particolare strutture di directory, che abbiamo già avuto modo di vedere ma che, per completezza, riportiamo anche nella Figura 6.1.

Questi file, al loro interno, devono conciliare l'esistenza sia del markup sia del codice tramite cui elaborare il contenuto del model stesso, e pertanto hanno un'estensione particolare (`.cshtml`) e sono scritti in una sintassi specifica che prende il nome dal **view engine** che sarà poi in grado di processarli, ossia **Razor**. Inizialmente introdotto con ASP.NET MVC, è disponibile in una versione aggiornata (e potenziata, come vedremo) per ASP.NET Core MVC.

Si tratta a tutti gli effetti di un nuovo linguaggio, per cui cercheremo di partire dalla sintassi di base per poi arrivare a realizzare pagine via via più complesse.

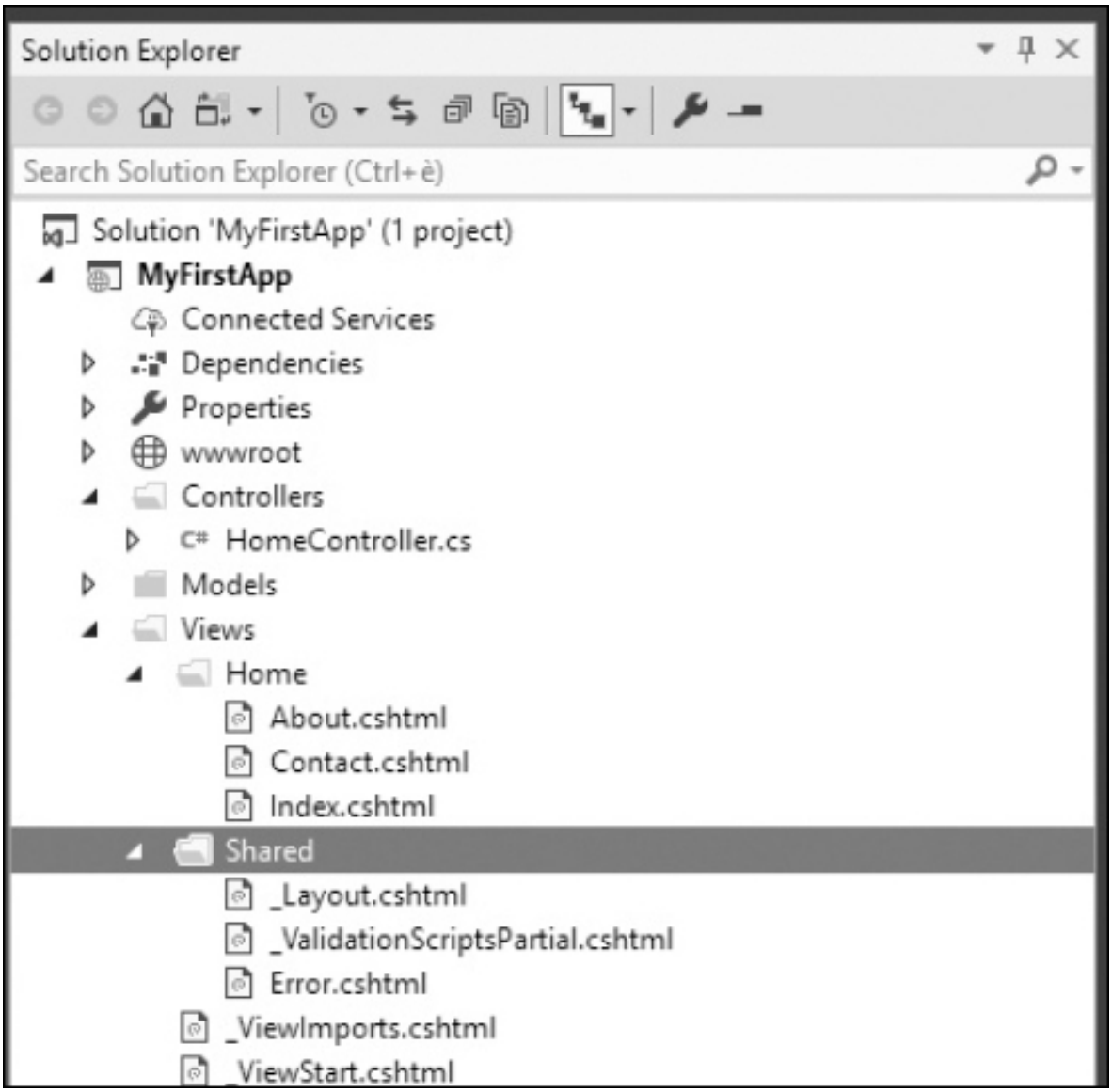


Figura 6.1 – Struttura delle directory per le view di un progetto ASP.NET Core MVC.

La sintassi di base

Visto che all'interno di una view devono coesistere sia codice sia markup, il compito di un view engine come Razor è innanzitutto quello di poter contraddistinguere questi due contesti, così che possiamo descrivere la logica secondo cui la nostra pagina dovrà essere generata. Il passaggio dal contesto di markup al contesto del codice (o viceversa) prende il nome di context switching e, nel caso di Razor, avviene grazie al

carattere @ per passare da markup a codice e tag (o @:) per passare da codice a markup.

In particolare, in una view, possiamo definire dei blocchi di codice tramite le parole chiave @{ ... } in C#, come viene mostrato nell'Esempio 6.1.

Esempio 6.1

```
@{
  ViewBag.Title = ″Index″;
  var myString = ″Una stringa di esempio″;
  int someValue = 24;
}
<div>
  <h1>Titolo pagina</h1>
  <p>Lorem ipsum dolor sit amet...</p>
</div>
@ {
  //qui C#
  <div>Qui markup</div>
  @: questo è ancora markup
  //qui di nuovo C#
}
```

Questi blocchi sono utili perché ci permettono di dichiarare delle variabili che poi saranno visibili all'interno del codice dell'intera view.

Di fianco al codice, ovviamente, possiamo includere anche del markup HTML, come possiamo notare nell'esempio in alto, senza dover prendere alcun accorgimento particolare. All'interno del markup, poi, possiamo effettuare all'occorrenza un nuovo context switching, ancora una volta utilizzando il carattere @; fintanto che restiamo all'interno di una singola riga, non è necessario che creiamo un nuovo blocco e possiamo sfruttare la sintassi inline dell'Esempio 6.2.

Esempio 6.2

```
<p>Oggi è @DateTime.Today</p>
<p>Indirizzo email: daniele@aspitalia.com</p>
<p>Seguimi su Twitter: @@dbochicchio</p>
<p style=″font-size:@(someValue)px″>Testo grande</p>
@*Questo è un commento*@
```

L'engine è abbastanza scaltro da distinguere i casi in cui il carattere `@` è inserito per altre finalità, come nel caso di un indirizzo email. Ovviamente non è detto che il context switching avvenga necessariamente nel contenuto di un tag. La terza riga del codice precedente mostra come possiamo sfruttare la variabile che abbiamo definito nell'Esempio 6.1 come componente dell'attributo style. Se necessario, possiamo isolare esplicitamente il blocco di codice che vogliamo passare a Razor, sfruttando le parentesi tonde, come abbiamo fatto nel codice precedente per separare il testo *"px"* dal nome della variabile `someValue`. L'uso invece di `@* ... *@` ci permette di inserire commenti nel codice C#, che non appariranno nel markup, dato che sono commenti server-side.

Branch e cicli

Tipicamente, quando realizziamo una view, abbiamo anche la necessità di visualizzare delle porzioni di pagina al verificarsi di una determinata condizione, o di ripetere lo stesso markup più volte. Per queste necessità possiamo sfruttare i blocchi `if`, `for` e `foreach`, o qualsiasi altro statement, come nell'Esempio 6.3.

Esempio 6.3

```
@if (DateTime.Today.DayOfWeek == DayOfWeek.Sunday)
{
  <div>Buona domenica</div>
}
else
{
  <p>Oggi è @DateTime.Today</p>
}
<ul>
  @for (int index = 0; index < 7; index++)
  {
    <li>@((DayOfWeek)index)</li>
  }
</ul>
```

La sintassi da utilizzare è piuttosto intuitiva, e sfrutta le caratteristiche di C#, con Razor che è in grado di individuare autonomamente i vari tag e considerarli pertanto come tali. Quando non abbiamo tag da inserire, possiamo sfruttare il tag speciale `<text>` dell'Esempio 6.4.

Esempio 6.4

```
@if (DateTime.Today.DayOfWeek == DayOfWeek.Sunday)
{
  <text>Buona domenica</text>
}
```

Questo tag viene ignorato in fase di rendering, ma serve a segnalare un contesto di markup al parser di Razor. Differentemente, il codice verrebbe considerato un'istruzione lato server (dopo la prima parantesi graffa) e pertanto avremmo un errore a runtime, poiché la stringa che segue non è compilabile in C#.

Definire funzioni in una view

Abbiamo già visto che, tramite il blocco `@{ ... }` possiamo dichiarare variabili che saranno visibili all'intera view. Alle volte può essere utile anche definire delle funzioni, e per questa necessità si usa il blocco `@functions` dell'Esempio 6.5.

Esempio 6.5

```
@functions {
    private string FormatDate(DateTime source)
    {
      return source.ToShortDateString();
    }
}
<p>Oggi è @FormatDate(DateTime.Today)</p>
```

Analogamente al caso delle variabili, anche queste funzioni saranno visibili all'interno della view e quindi possono essere utili per centralizzare parte della logica e riutilizzarla in più punti.

Le view e il model: tipizzazione debole e forte

In tutti gli esempi che abbiamo realizzato fino a questo punto del libro, il model ha avuto un'importanza davvero marginale: tutte le volte che abbiamo avuto la necessità di passare informazioni dal controller alla view, abbiamo sfruttato l'oggetto `ViewBag` (o `ViewData`), ossia un `Dictionary` condiviso tra questi due componenti del pattern MVC.

In realtà, questo modo di procedere è davvero poco pratico nel momento in cui ci troviamo a realizzare applicazioni reali: usando questi strumenti, infatti, non abbiamo alcun ausilio dal tool di sviluppo, né dal punto di vista della correttezza delle chiavi che stiamo utilizzando, né dal punto di vista della tipizzazione. Consideriamo l'Esempio 6.6.

Esempio 6.6

```
public IActionResult Wrong()
{
  ViewBag.Today = ″non è una data″;
  return View();
}
```

Esempio 6.6 – View

```
<h2>@ViewBag.Today.ToShortDateString()</h2>
```

Sebbene il codice precedente sia palesemente errato, Visual Studio non è in grado di darci alcun tipo di avviso durante la scrittura, proprio in virtù del fatto che stiamo sfruttando il late binding, ossia stiamo rimandando al runtime il controllo della congruità dei tipi.

Il risultato è che, se proviamo a visualizzare la pagina, otteniamo l'errore di runtime mostrato nella Figura 6.2.

An unhandled exception occurred while processing the request.

RuntimeBinderException: 'string' does not contain a definition for 'ToShortDateString'

CallSite.Target(Closure , CallSite , object)

Stack Query Cookies Headers

RuntimeBinderException: 'string' does not contain a definition for 'ToShortDateString'

```
CallSite.Target(Closure , CallSite , object )
System.Dynamic.UpdateDelegates.UpdateAndExecute1<T0, TRet>(CallSite site, T0 arg0)
AspNetCore._Views_Home_Wrong_cshtml+<ExecuteAsync>d__0.MoveNext() in Wrong.cshtml
+       6. <h2>@ViewBag.Today.ToShortDateString()</h2>
System.Runtime.ExceptionServices.ExceptionDispatchInfo.Throw()
System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)
Microsoft.AspNetCore.Mvc.Razor.RazorView+<RenderPageCoreAsync>d__16.MoveNext()
System.Runtime.ExceptionServices.ExceptionDispatchInfo.Throw()
System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)
Microsoft.AspNetCore.Mvc.Razor.RazorView+<RenderPageAsync>d__15.MoveNext()
System.Runtime.ExceptionServices.ExceptionDispatchInfo.Throw()
System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)
System.Runtime.CompilerServices.TaskAwaiter.GetResult()
Microsoft.AspNetCore.Mvc.Razor.RazorView+<RenderAsync>d__14.MoveNext()
System.Runtime.ExceptionServices.ExceptionDispatchInfo.Throw()
System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)
Microsoft.AspNetCore.Mvc.ViewFeatures.ViewExecutor+<ExecuteAsync>d__22.MoveNext()
System.Runtime.ExceptionServices.ExceptionDispatchInfo.Throw()
System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)
Microsoft.AspNetCore.Mvc.ViewFeatures.ViewExecutor+<ExecuteAsync>d__21.MoveNext()
System.Runtime.ExceptionServices.ExceptionDispatchInfo.Throw()
System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)
Microsoft.AspNetCore.Mvc.ViewResult+<ExecuteResultAsync>d__26.MoveNext()
System.Runtime.ExceptionServices.ExceptionDispatchInfo.Throw()
System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)
```

Figura 6.2 – Errore a runtime dovuto alla mancata tipizzazione.

La soluzione sicuramente più manutenibile e sicura è quella di realizzare un model, che mantenga al suo interno tutte le informazioni da visualizzare, in maniera tipizzata. ASP.NET Core MVC non pone particolari vincoli alle loro realizzazione, anche se per convenzione è preferibile inserirli nella directory `Models` e denominarli con il suffisso *-ViewModel*. Il model per la home page è quello dell'Esempio 6.7.

Esempio 6.7

```
public class HomeViewModel
{
  public string Title { get; set; }
  public DateTime TheDate { get; set; }
}
```

A questo punto non dobbiamo far altro che creare un'istanza di questa classe all'interno della action e passarla come parametro alla view, come nell'Esempio 6.8.

Esempio 6.8

```
public IActionResult TypedAction()
{
  var model = new HomeViewModel()
  {
    Title = "Action tipizzata",
    TheDate = DateTime.Today
  };
  return View(model);
}
```

Affinché possiamo sfruttare `HomeViewModel` anche nel codice della view, dobbiamo creare quella che, in ASP.NET Core MVC, prende il nome di view tipizzata, grazie all'apposito checkbox della finestra di dialogo della Figura 6.3. In particolare, Visual Studio ci permette anche di selezionare il tipo che vogliamo utilizzare tramite una combobox.

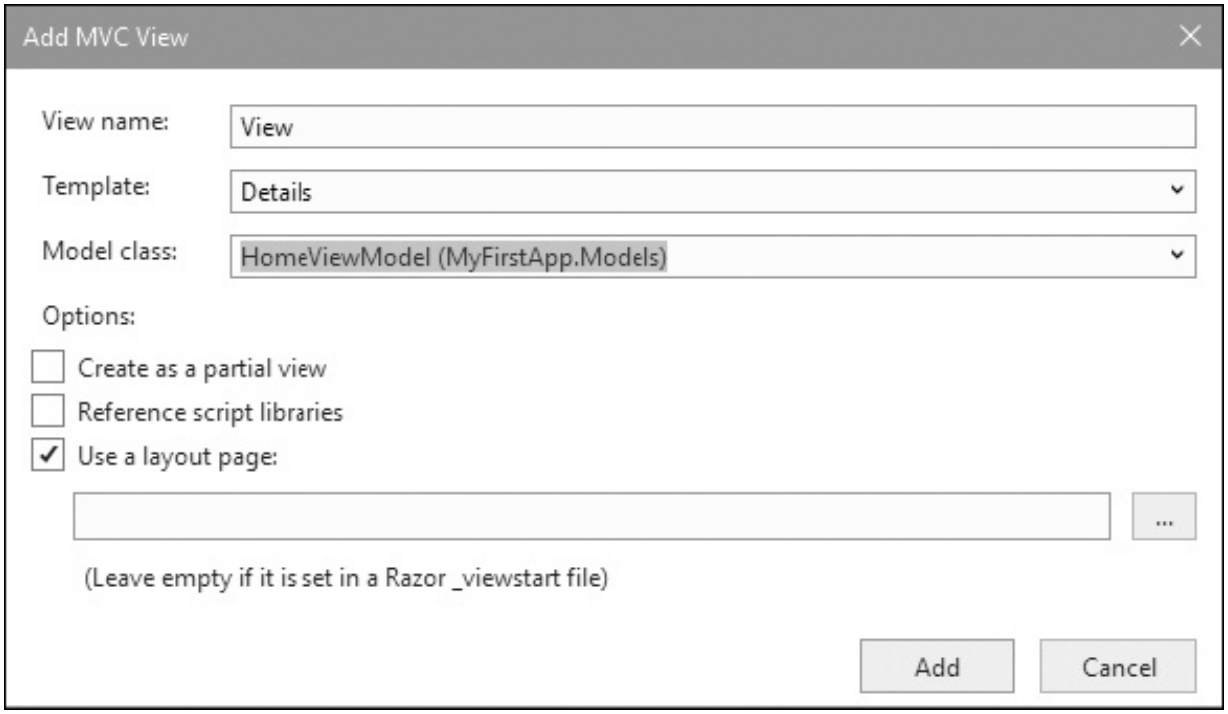


Figura 6.3 – Finestra di dialogo per creare una view tipizzata.

Il risultato di questa operazione è una nuova view che, a differenza delle precedenti che abbiamo creato finora, presenta la direttiva `@model` che indica il tipo di model utilizzato, come nell'Esempio 6.9.

Esempio 6.9

```
@model HomeViewModel
@{
  ViewBag.Title = ″TypedAction″;
}
<h2>@Model.Title</h2>
```

Il vantaggio di questo approccio è che ci ritroviamo a disposizione una proprietà `Model` che, in virtù di questa direttiva, è di tipo `HomeViewModel`, e quindi ci permette di accedere direttamente alle proprietà; in questo modo Visual Studio è in grado di accorgersi preventivamente se abbiamo sfruttato proprietà non valide o di fornirci supporto al momento della scrittura del codice tramite l'Intellisense, come possiamo vedere nella Figura 6.4.

Figura 6.4 – Intellisense durante la scrittura di una view.

ASP.NET Core MVC supporta una modalità di pre-compilazione delle view in fase di pubblicazione che può aiutarci a non commettere errori nelle view (ad esempio sbagliando a specificare il nome di una proprietà) e che si può attivare o disattivare agendo sul nodo `MvcRazorCompileOnPublish` all'interno del file `.csproj` (di default è attivata).

L'Esempio 6.10 mostra come attivare la funzionalità, che ha bisogno che sia referenziato il package `Microsoft.AspNetCore.Mvc.Razor.ViewCompilation` (solo se il target è .NET Framework, per .NET Core non è necessario), oppure ai metapackage `Microsoft.AspNetCore.All` (2.0) e `Microsoft.AspNetCore.All` (2.1).

Esempio 6.10

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
    <MvcRazorCompileOnPublish>true</MvcRazorCompileOnPublish>
  </PropertyGroup>
…
</Project>
```

Questa impostazione ha l'effetto collaterale di rendere leggermente più lenta la compilazione, soprattutto se abbiamo molte view nel progetto, ma ha il vantaggio di segnalare già a compile time eventuali incongruenze, come viene mostrato nella Figura 6.5.

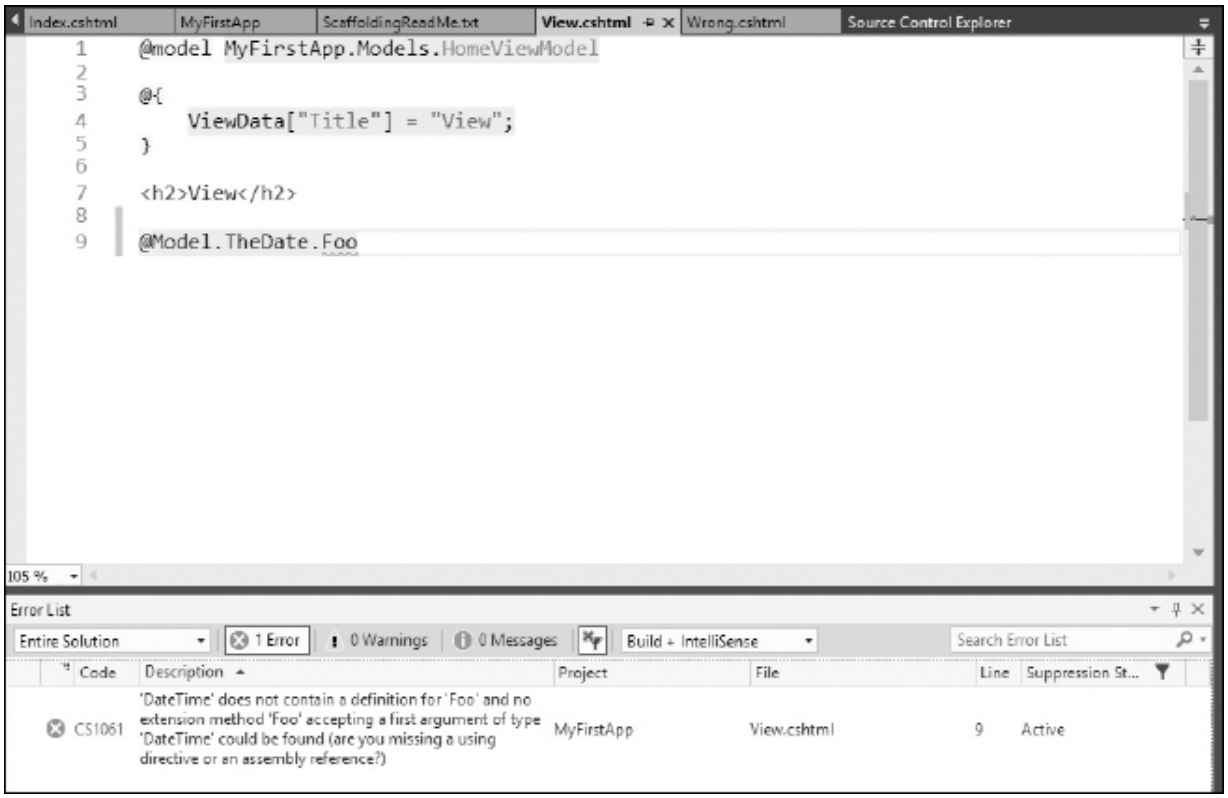


Figura 6.5 – Errore in compilazione di una view.

Grazie a questa funzionalità ASP.NET Core MVC provvederà a pre-compilare le view e, a differenza di ASP.NET MVC, le stesse non dovranno essere compilate da Razor in C# in fase di esecuzione, con benefici anche in termini di semplicità di deployment (non ci sono tanti file da distribuire) e cold start (l'avvio dell'applicazione a freddo), poiché i tempi si riducono e le performance migliorano nettamente.

Le view, per default, sfruttano i namespace specificati all'interno del file `_ViewImports.cshtml`, posto all'interno della directory `Views` del progetto, a differenza di ASP.NET MVC, che lo fa all'interno del `web.config` (che qui manca). Se abbiamo bisogno di utilizzarne altri ancora, possiamo modificare questo file, o importarli sistematicamente all'interno di ciascuna view, tramite la direttiva `@using`. L'Esempio 6.11 mostra entrambe queste soluzioni (la differenza è solo il file in cui vengono effettuate). Di default il template di progetto importa il namespace dell'applicazione e quello del suo model.

Esempio 6.11

```
@using MyApplication
@using MyApplication.Models
```

Con ASP.NET Core 2.1 è stata anche aggiunta la possibilità di distribuire una class library con all'interno blocchi di UI compilata a partire da file Razor. Avremo modo di tornare su questa funzionalità nel Capitolo 15.

A questo punto abbiamo sicuramente capito come l'uso di view fortemente tipizzate semplifichi di molto tutto il processo di sviluppo e abbia un peso fondamentale nell'abilità di accorgersi di eventuali problemi, prima che questi generino eccezioni. Si tratta di un requisito fondamentale per le applicazioni reali, ma di certo non è il solo. Una lacuna che dobbiamo colmare, per esempio, è la modalità secondo cui possiamo assicurare al nostro sito una consistenza grafica tra le varie pagine. Sarà l'argomento della prossima sessione.

Consistenza grafica tra le pagine: la layout view

Le layout page rappresentano uno strumento assolutamente indispensabile per mantenere uniforme il look delle pagine del nostro sito web. Tramite le layout page, infatti, possiamo definire un layout di base e un insieme di elementi di base, unitamente a una serie di placeholder che, nelle singole pagine interne, possiamo popolare con il contenuto specifico della pagina che l'utente sta visualizzando.

Le layout view sono delle normali view e pertanto non c'è una particolare tipologia di file da utilizzare: per aggiungerne una al nostro progetto non dobbiamo far altro che creare una nuova view, tipicamente all'interno della directory `Shared`, visto che con ogni probabilità dovrà essere accessibile da molteplici controller.

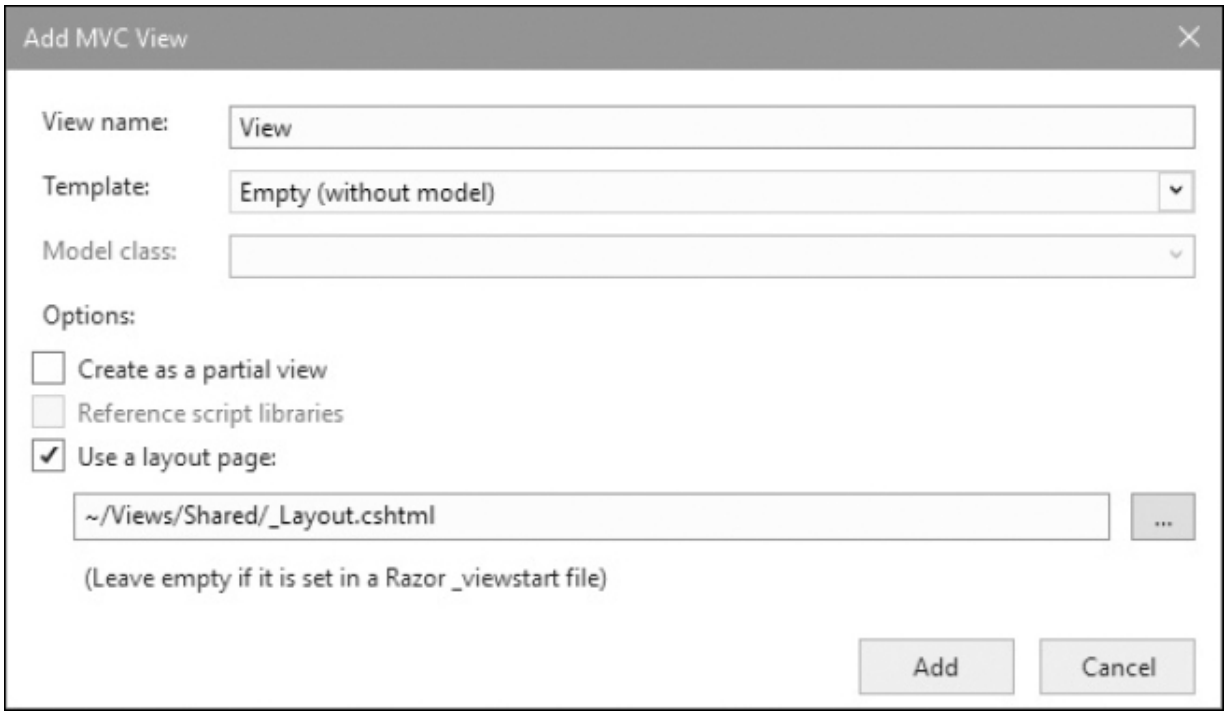


Figura 6.6 – Aggiunta di una layout view.

Come possiamo notare nella Figura 6.6, ci sono un paio di regole non scritte che vengono di solito adottate quando si crea una view di questo tipo.

- Il nome utilizzato è `_Layout.cshtml`; per convenzione, infatti, in ASP.NET Core MVC si preferisce utilizzare il carattere underscore come prefisso di tutte le view condivise.

- Non è una view tipizzata, perché sarà utilizzata in un gran numero di situazioni e vogliamo lasciare alle singole action la flessibilità di utilizzare il model più consono alla view che dovrà essere mostrata.

Sfruttare le layout view nel progetto

Dal punto di vista del codice, invece, non c'è nulla di nuovo rispetto alle regole viste finora, se non per la presenza, al suo interno, della chiamata al metodo `RenderBody`. L'Esempio 6.12 ci mostra una semplicissima layout view.

Esempio 6.12

```
<!DOCTYPE html>
<html>
  <head>
    ...
  </head>
  <body>
    <div id="body">
      @RenderBody()
    </div>
  </body>
</html>
```

Questo metodo rappresenta il segnaposto in cui, effettivamente, verrà renderizzato il contenuto specifico della pagina richiesta: ogni view ha una proprietà `Layout` che serve allo stesso scopo, e che dobbiamo valorizzare con l'URL della layout view desiderata.

Esempio 6.13

```
@model HomeViewModel
@{
  Layout = "~/Views/Shared/_Layout.cshtml";
}
<h2>Contenuto specifico della pagina</h2>
```

Una cosa che dobbiamo dire è che questa proprietà può essere liberamente valorizzata anche a runtime, prima della fase di execute della view.

> *ASP.NET Core MVC supporta anche layout view innestate; come abbiamo avuto modo di specificare, una layout view è a tutti gli effetti una normalissima view e pertanto nulla ci vieta di impostarne la proprietà Layout in modo che punti a un'ulteriore layout view.*

Generalmente, si opta per una registrazione globale della layout view, come vedremo nel prossimo paragrafo di questo capitolo.

La view _ViewStart

Impostare la proprietà `Layout` per ogni view della nostra applicazione può essere sicuramente tedioso e poco semplice da manutenere, nel momento in cui, in futuro, dovessimo decidere di modificare questi riferimenti. Per questa ragione, ASP.NET Core MVC mette a disposizione uno strumento, ossia una view speciale, denominata `_ViewStart`.

Si tratta di un file di Razor, all'interno del quale possiamo definire del codice, e che ha la caratteristica di essere eseguito, in maniera del tutto automatica, prima del rendering di una qualsiasi view. Questo ci consente di specificare all'interno di `_ViewStart` quale sarà la layout view che sarà adottata per default da tutte le pagine dell'applicazione, senza che dobbiamo valorizzarla all'interno di ogni file, come nell'Esempio 6.14.

Esempio 6.14

```
@{
  Layout = "~/Views/Shared/_Layout.cshtml";
}
```

Se diamo una nuova occhiata alla Figura 6.1, presente all'inizio di questo capitolo, possiamo notare che, tipicamente, il file `_ViewStart` viene creato direttamente all'interno della directory *Views*. In realtà, in un progetto possono coesistere molteplici file `_ViewStart`, ognuno in una specifica directory. Il risultato è che il runtime li eseguirà in sequenza, partendo da quello più esterno fino a quello più specifico, contenuto nella directory della view selezionata. Nella Figura 6.7, per esempio, abbiamo creato un file `_ViewStart` all'interno della directory `Home`.

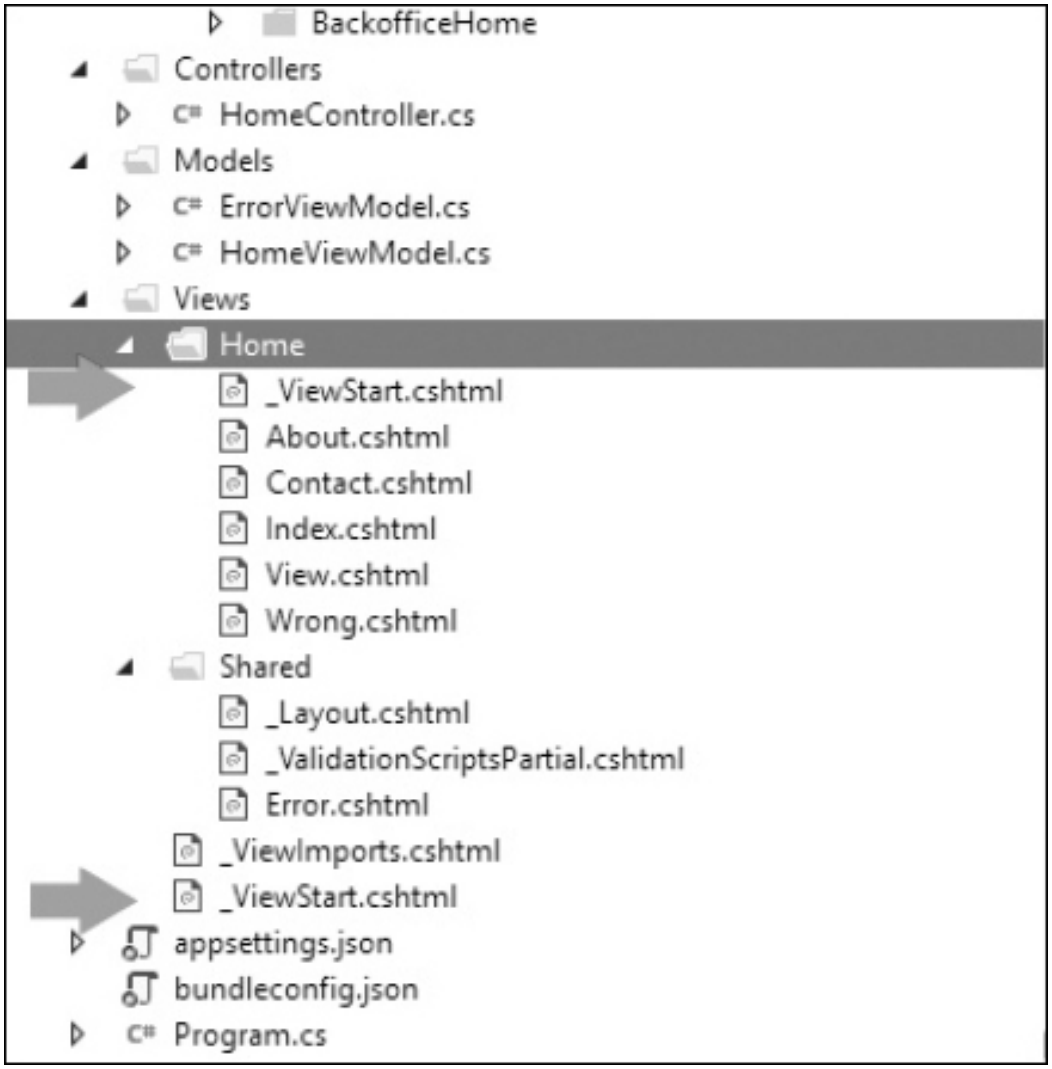


Figura 6.7 – Una struttura di directory con _ViewStart dentro home.

Se a questo punto proviamo a invocare l'action `Index`, verranno eseguiti, nell'ordine:

- Il file `_ViewStart.cshtml` contenuto nella directory `Views`.
- Il file `_ViewStart.cshtml` contenuto nella directory `Home`.

Questo ci permette, per esempio, di specificare una view di layout differenti per aree funzionali del sito, o addirittura sul singolo controller.

Com'è lecito attendersi, in ogni caso, l'ultima parola spetta sempre alla view che verrà renderizzata, che a sua volta potrà indicare una

layout view specifica.

Definire sezioni aggiuntive in una layout view

Oltre al contenuto di default, che viene incluso tramite il metodo `RenderBody`, una content view può definire anche delle sezioni aggiuntive, ognuna **identificata da un nome**, tramite la direttiva `@section`, come nell'Esempio 6.15.

Esempio 6.15

```
@section Footer {
  <p>Footer della content view</p>
}
```

Questi blocchi vengono ignorati da `RenderBody`, ma richiedono un metodo `RenderSection` all'interno della layout view, per specificare dove devono essere posizionati.

Esempio 6.16

```
<div id="body">
  @RenderBody()
</div>

<div id="footer">
  @RenderSection("Footer")
  @RenderSection("OptionalSection", required:false)
</div>
```

Per default, ogni sezione aggiuntiva deve essere presente nella content view, altrimenti verrà sollevato un errore a runtime, a meno che non specifichiamo il contrario con il parametro `required`, come abbiamo fatto nell'Esempio 6.16 per *OptionalSection*.

L'alternativa è sfruttare il metodo `IsSectionDefined` per scoprire se una sezione è definita nella content view, come nell'Esempio 6.17.

Esempio 6.17

```
<div id="footer">
  @if (IsSectionDefined("Footer"))
  {
    @RenderSection("Footer")
  }
  else
  {
    <p>Contenuto di default definito su layout view</p>
  }
  @RenderSection("OptionalSection", required: false)
</div>
```

Questa tecnica, come possiamo vedere nel codice precedente, può essere utilizzata anche per produrre un contenuto di default.

Le layout page, insomma, sono uno strumento estremamente potente, grazie al quale possiamo realizzare un sito web che si visualizzi correttamente su ogni dispositivo utilizzato, abbinandolo all'uso di file CSS che facciano uso di tecniche come il responsive design, che di default nel template predefinito viene attuato grazie all'utilizzo di Bootstrap. Al momento, però, siamo ancora costretti a scrivere manualmente la totalità dell'HTML della pagina. Uno strumento messo a disposizione da ASP.NET Core MVC per rendere più semplice la scrittura del codice di una view è rappresentato dagli HTML e dai tag helper. Ne parleremo nella prossima sezione.

Semplificare il codice delle view: gli HTML e i tag helper

Se abbiamo già lavorato con ASP.NET Web Forms, sicuramente uno degli aspetti che, al primo impatto, sembrano più ostici nell'approccio ad ASP.NET Core MVC è l'assenza di controlli server side evoluti: quando realizziamo le view, infatti, siamo costretti a "sporcarci le mani" con il vecchio codice HTML, mentre in realtà ci piacerebbe lavorare a un livello più alto di astrazione dei semplici tag.

Questi tipi di necessità sono gestite in ASP.NET Core MVC tramite gli **HTML helper**, ossia dei metodi che possiamo sfruttare per produrre in maniera automatica dei blocchi di markup, anche complessi. Mutuati da ASP.NET MVC, gli HTML Helper in ASP.NET Core MVC sono affiancati

anche dai Tag helper, che ne semplificano ulteriormente le potenzialità e che ricordano più da vicino i controlli di ASP.NET Web Forms.

Analizzeremo prima gli HTML helper e poi passeremo, invece, ad analizzare i Tag helper.

Gli HTML Helper

Dal punto di vista strettamente pratico, si tratta di extension method della classe `HtmlHelper`, che è esposta dalla view tramite la proprietà `Html`. Come possiamo notare nella Figura 6.8, ASP.NET Core MVC definisce diversi metodi di questo tipo, per la maggior parte sfruttabili per la creazione di elementi utili per le form, quali `TextBox`, `ListBox` o `RadioButton`, solo per citarne alcuni.

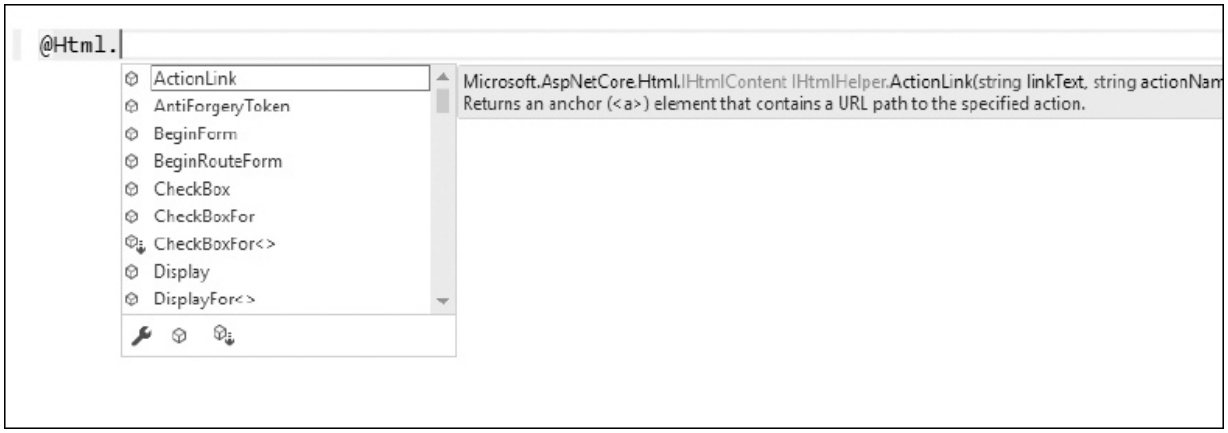


Figura 6.8 – HTML helper standard di ASP.NET Core MVC.

Daremo una rapida occhiata a questi argomenti, per poi spostarci subito ai Tag helper, che sono molto più interessanti.

Il metodo ActionLink

Quando dobbiamo inserire un link verso un'altra pagina del nostro sito, l'utilizzo diretto di un URL scritto staticamente in un tag <a> non è consigliabile, perché ci vincola a specificare l'URL di destinazione in maniera prefissata nel codice della view mentre, come abbiamo visto finora, in generale questo è il prodotto della configurazione del routing.

Una strada più semplice è sfruttare l'HTML helper `ActionLink`, tramite il quale possiamo specificare la destinazione nei termini di controller e action. L'Esempio 6.18 mostra alcuni casi di utilizzo.

Esempio 6.18

```
@* Link alla action SomeAction dello stesso controller *@
@Html.ActionLink("Semplice", "SomeAction")

@* Link alla action Index di CustomersController *@
@* In virtù dei default, l´URL sarà /Customers *@
@Html.ActionLink("Con controller", "Index", "Customers")

@* genera <a href="../" target="_blank"> *@
@Html.ActionLink("Attributi personalizzati", "Index",
  "Customers", null, new {
    target = "_blank", style = "font-size:20px" })

@* Genera un link all´URL /Customers/Detail/23 con la classe css
   myLink *@
@Html.ActionLink("Link con parametri", "Detail", "Customers",
  new { id = 23 }, new { @class = "myLink" })
```

Come possiamo notare nel codice precedente, esistono diversi overload che possiamo sfruttare per generare varie tipologie di URL e personalizzare il markup generato. In particolare, per indicare i parametri addizionali o attributi HTML, abbiamo utilizzato un anonymous type; esiste anche un overload che invece sfrutta un `RouteValueDictionary`, ma la sintassi dell'esempio è sicuramente più comoda.

Questo metodo, oltre al vantaggio di rendere indubbiamente più semplice la determinazione dell'URL, ha anche la peculiarità di adattarsi alla configurazione del routing: se in futuro dovessimo decidere di modificare queste impostazioni, infatti, la consistenza dei link del nostro sito web rimarrebbe comunque assicurata.

In alcune occasioni, per esempio se stiamo scrivendo del codice JavaScript, potremmo avere la necessità di recuperare solo l'URL di una determinata route, piuttosto che generare un vero e proprio link. In questo caso possiamo sfruttare la classe `UrlHelper` come nell'Esempio 6.19.

Esempio 6.19

```
<script type="text/javascript">
  function myFunction() {
    var url =
      '@Url.Action("Detail", "Customers", new { id = 23 })';
    window.open(url);
  }
</script>
```

Il risultato del metodo `Action` è una stringa contenente l'URL desiderato, che nel codice precedente abbiamo usato per popolare la variabile `url`.

Il metodo RouteLink

L'helper `ActionLink` che abbiamo visto nella sezione precedente è sicuramente molto versatile, ma ha il limite di funzionare esclusivamente con route di ASP.NET Core MVC. Nell'ambito di applicazioni complesse, che magari sfruttano molteplici tecnologie contemporaneamente, non è detto che siano definite solo route di questo tipo. Nel Capitolo 4, infatti, abbiamo visto che in generale una route può definire diversi parametri e gestire la chiamata con un qualsiasi `IRouteHandler`; il fatto di avere parametri nella route quali controller e action, insomma, è solo un caso particolare, per quanto comune.

Se abbiamo bisogno di lavorare più a basso livello, possiamo sfruttare l'HTML helper `RouteLink`. Immaginiamo di aver definito, nella classe `RouteConfig`, la route dell'Esempio 6.20.

Esempio 6.20

```
routes.Add("CustomRoute",
  new Route("Custom/{first}/{second}/{third}",
  new CustomRouteHandler()));
```

`ActionLink` non è in grado di generare un link a questa route, perché non contiene nozioni di controller o action. In questo caso, allora, dobbiamo sfruttare `RouteLink` come nell'Esempio 6.21, che ci permette di specificare la route e indicare puntualmente i `routeValues` desiderati per determinare l'URL.

Esempio 6.21

```
@*Genera un link a /Custom/First/Second/Third*@
@Html.RouteLink(″Link a customRoute″, ″customRoute″,
  new { first = ″First″, second = ″Second″, third = ″Third″ })
```

Anche in questo caso, se invece di costruire un link dobbiamo semplicemente determinare l'URL, possiamo sfruttare il metodo `Url.RouteUrl`.

Il metodo Raw

Da ciò che abbiamo appreso finora, per includere in pagina un contenuto variabile tutto ciò che dobbiamo fare è esporre il dato tramite una proprietà del model e referenziarlo all'interno del markup, come nell'Esempio 6.22.

Esempio 6.22

```
<p>@Model.SomeProperty</p>
```

In linea generale, visualizzare in pagina contenuto variabile, che magari proviene dal database o da un precedente input dell'utente, è un'operazione che comporta un certo grado di rischio, visto che potremmo essere esposti ad attacchi di tipo XSS, che approfondiremo nel Capitolo 17. In realtà, finora non abbiamo mai messo in luce l'argomento perché, per ovvie ragioni di sicurezza, Razor effettua automaticamente l'encoding del testo.

Esistono dei casi, tuttavia, in cui vogliamo disabilitare l'encoding, magari perché `SomeProperty` contiene del testo formattato, che altrimenti non verrebbe visualizzato correttamente. In questi casi possiamo usare l'helper `Raw` dell'Esempio 6.23.

Esempio 6.23

```
<p>@Html.Raw(Model.SomeProperty)</p>
```

Per le ragioni che abbiamo citato, però, dobbiamo essere molto attenti quando usiamo questo metodo e prendere le dovute precauzioni, magari rimuovendo da `SomeProperty` eventuali tag potenzialmente dannosi, come `<script>`, `<iframe>` e via discorrendo. In generale, se vogliamo visualizzare una stringa a video senza che Razor ne faccia l'encoding, sarà sufficiente utilizzare il tipo `HtmlString`, in luogo di `string`.

Il metodo Partial e le partial view

Le layout page che abbiamo introdotto nel corso di questo capitolo svolgono sicuramente una funzione fondamentale nell'ottica di mantenere costante il look del nostro sito web tra le diverse pagine. Purtroppo, però, da sole non sono sufficienti a risolvere il problema nella sua interezza. Spesso, infatti, ci troviamo nella necessità di replicare più volte, in diverse view, alcuni blocchi di contenuto; per i casi semplici abbiamo a disposizione gli HTML helper, ma quando il markup diviene complesso e le variabili in gioco sono molteplici, abbiamo bisogno di uno strumento più versatile: le **partial view**.

Per questa tipologia di view, valgono le stesse regole che abbiamo illustrato fino a questo momento, a partire dalla modalità di creazione, che avviene tramite la solita finestra di dialogo della Figura 6.9, che abbiamo visto più volte nel corso di questo capitolo. L'unica differenza rispetto ai casi precedenti è rappresentata dalla spunta sulla voce *Create as partial view*.

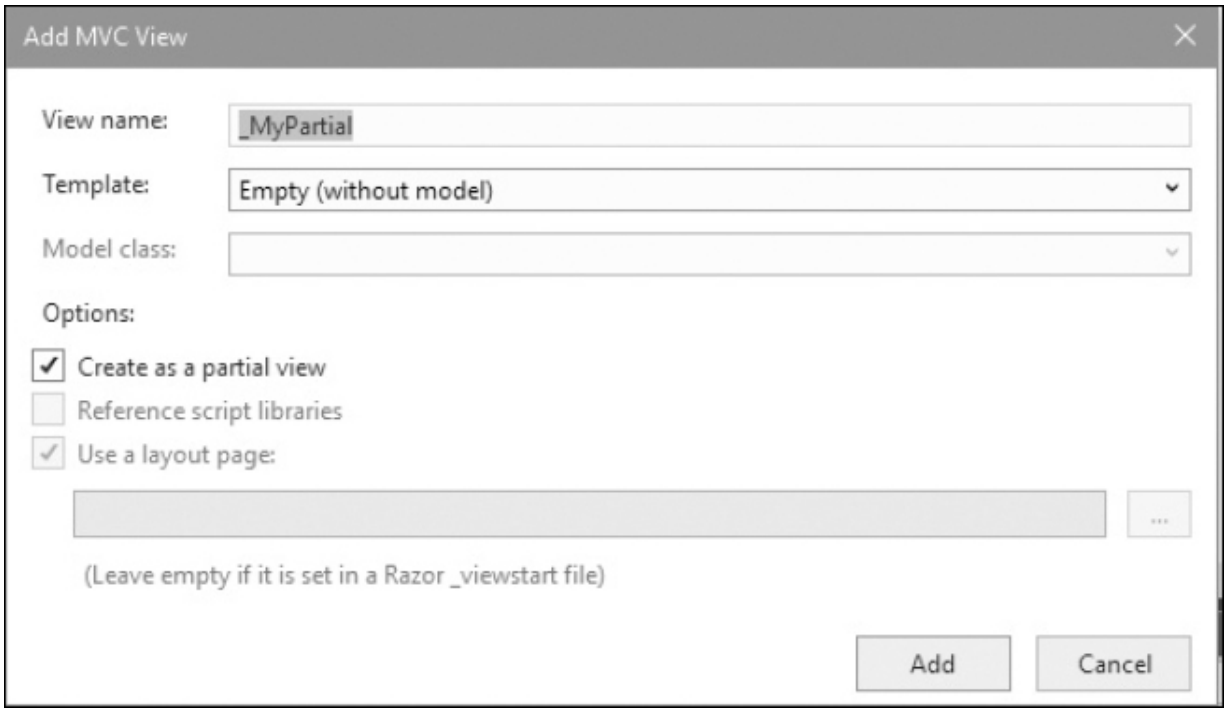


Figura 6.9 – Creazione di una partial view.

Come possiamo notare nella figura, le partial view possono essere tipizzate o non tipizzate e, ovviamente, non hanno mai un riferimento a una layout view, visto che non sono della pagine autonome, ma vanno inserite all'interno di una content view. Dal punto di vista del codice, non c'è nulla di nuovo rispetto a quanto abbiamo visto finora, come possiamo notare nell'Esempio 6.24, relativo a una partial view tipizzata.

Esempio 6.24

```
@model HomeViewModel
<p>Questo paragrafo appartiene alla partial view</p>
<p>@Model.Title</p>
```

Per visualizzare una partial view all'interno della pagina, dobbiamo sfruttare l'HTML helper `Partial`, come nell'Esempio 6.25.

Esempio 6.25

```
@Html.Partial("MyPartialView", Model)
```

Questo metodo accetta come parametro principale il nome della view; nel caso del codice precedente, ASP.NET cercherà un file di nome `MyPartialView.cshtml` all'interno della directory del controller in esecuzione, o in Views\Shared, dove dobbiamo posizionare il file nel caso in cui vogliamo che possa essere referenziato da diversi controller. Nell'Esempio 6.25, inoltre, abbiamo sfruttato un particolare overload per passare anche un'istanza del model, visto che la nostra partial view ne ha bisogno.

I Tag helper e la gestione del markup

Come abbiamo visto, lo scopo per cui sono stati introdotti gli HTML helper è quello di semplificare la creazione del markup, essendo di fatto delle classi particolari che si incastrano con l'infrastruttura di rendering di Razor.

Tutto quello che scriviamo è, infatti, preso da Razor e convertito in una classe C#, con chiamate che ricostruiscono il codice che abbiamo scritto e che, a runtime, genereranno l'HTML corrispondente.

Un concetto caro a chi ha sviluppato con ASP.NET Web Forms è quello dei componenti, cioè di markup che viene inserito nella pagina e poi viene programmato, evitando di ragionare in funzione di solo codice: quest'ultimo, infatti, è sempre soggetto ad errori ed è scarsamente apprezzato da web designer o sviluppatori front-end che lavorano ad una view, a cui bisogna insegnare specificatamente come agire. Per questo motivo, sono nati i Tag helper, che rappresentano pezzi di markup inseriti nella pagina e che poi a runtime produrranno un output specifico, esattamente come gli HTML Helper.

Di fatto, rappresentano l'anello di giunzione tra HTML helper e HTML e non vanno a sostituirsi ai primi: Tag helper e HTML helper convivono serenamente. Non è detto che ci sia un Tag helper per ogni HTML helper e viceversa. Ci sono casi in cui vale la pena adottare una scelta e casi in cui è meglio affidarsi a un'altra. Avremo modo di analizzare i principali

Tag helper subito e rimandare agli altri nel prossimo capitolo, perché, come vedremo, sono l'ideale proprio in presenza di form.

Il Tag helper Partial

Un tag helper minimale, introdotto con ASP.NET 2.1, può essere quello dell'Esempio 6.26, che consente di effettuare il rendering di una partial view.

Esempio 6.26

```
<partial name="_MyPartialView" asp-for="Model" />
```

Come si può notare, la chiamata precedente lascia posto ad un tag HTML, che poi Razor trasformerà nello stesso risultato, cioè quello di includere la view parziale. In questo caso, grazie all'attributo `name` possiamo specificare la view a cui fare riferimento, mentre con `asp-for` andiamo a passare il model. Tecnicamente, quello che facciamo con quest'ultima proprietà è specificare una `ModelExpression`, che poi andrà ad assegnare il modello, come nel caso dell'HTML helper. Con le stesso approccio è possibile passare anche un ViewData specifico, agendo sulla proprietà `view-data` del tag.

Rispetto all'utilizzo della sintassi basata su HTML helper, in questo caso il rendering è sempre effettuato in maniera asincrona.

Il Tag helper Environment

Un caso particolarmente interessante tra i Tag helper predefiniti è quello denominato `Environment`. Nei capitoli precedenti, in particolare nel Capitolo 3, abbiamo visto come ASP.NET Core supporti già il concetto di ambiente di esecuzione. Questo tag helper non fa altro che rendere possibile la differenziazione del markup emesso in funzione dell'ambiente e ritorna molto comodo in fase di sviluppo, per caricare file JavaScript o CSS non minified, lasciando questa eventualità solo alla produzione, piuttosto che per emettere markup specifico, evitando del

tutto di affidarsi a un blocco di codice con una condizione, che potrebbe più facilmente contenere errori.

Nell'Esempio 6.27 possiamo vedere come sfruttare questo Tag helper.

Esempio 6.27

```
<environment include="Staging,Production">
  <strong>Staging o Production</strong>
</environment>
<environment exclude="Staging">
  <strong>Production o Development</strong>
</environment>
```

Nell'esempio possiamo notare due utilizzi, legati agli attributi `include` e `exclude`, che servono rispettivamente a specificare (separati da virgola) gli ambienti per cui emettere quel markup, oppure quelli da escludere (sempre separati da virgola).

Non c'è molto altro da aggiungere, poiché, in caso la condizione non dovesse venire rispettata, nessun markup verrebbe emesso dal Tag helper.

Il Tag helper Anchor

Uno dei primi esempi di HTML helper che abbiamo introdotto è stato quello per generare link attraverso `ActionLink`. Questo rappresenta una delle cause per cui è più facile che uno sviluppatore alle prime armi commetta degli errori, poiché l'unico modo di passare i valori a questo metodo è per posizione. Il grande vantaggio dei Tag helper, invece, è quello di essere molto espressivi.

L'Esempio 6.18, prima introdotto, può essere "tradotto" con i tag riportati nell'esempio che segue.

Esempio 6.28

```
@* Link alla action SomeAction dello stesso controller *@
<a asp-action="SomeAction">Semplice</a>
@* Link alla action Index di CustomersController
    In virtù dei default, l´URL sarà /Customers *@
```

```
<a asp-controller="Customers"
  asp-action="Index">Con controller</a>
@* Genera <a href=".." target="_blank"> *@
<a asp-controller="Customers"
  asp-action="Index"
  target="_blank"
  style="font-size:20px">Attributi personalizzati</a>
@* Genera un link all´URL /Customers/Detail/23 con la classe css myLink *@
<a asp-controller="Customers"
  asp-action="Index"
  asp-route-id="23"
  class="myLink">Attributi personalizzati</a>
```

Per uno sviluppatore HTML (o che non ha mai visto ASP.NET MVC in precedenza) questa sintassi risulta molto più leggibile e gestibile. In realtà, è semplice da utilizzare per chiunque, poiché ci sono poche proprietà che regolano il modo in cui verrà generato l'URL finale.

- `asp-controller`: indica il controller da utilizzare e, se omesso, presuppone lo stesso controller abbinato alla action che ha renderizzato la view;
- `asp-action`: per indicare la action da invocare;
- `asp-route-{parametro}`: per indicare il parametro di routing da passare.

Tutti gli altri attributi, se non riconosciuti, verranno lasciati come sono e, pertanto, renderizzati nell'HTML risultante.

Analogamente, è possibile referenziare anche una route per nome, specificando solo l'attributo `asp-route`, o passare tutti i parametri di routing (sotto forma di `Dictionary`) grazie all'attributo asp-all-route-data, come si vede nell'Esempio 6.29.

Esempio 6.29

```
@* corrisponde ad una route che porta a /Profile/ *@
<a asp-route="MyProfile">Il mio profilo</a>
@* Passa tutti i parametri in un colpo solo alla route *@
@{
var parms = new Dictionary<string, string>
  {
    { "searchKey", "ASP.NET" },
```

```
      { ″author″, ″Daniele Bochicchio″ }
    };
  }
  <a asp-route=″SearchForBooks″
    asp-all-route-data=″parms″>Tutti i libri di Daniele Bochicchio
    su ASP.NET</a>
```

Per concludere, esiste un attributo `asp-fragment` che consente di passare un fragment all'URL, cioè quella parte che si trova nell'URL dopo il carattere # e che serve a navigare internamente ad un documento HTML.

Il Tag helper Image

Per concludere questa prima carrellata, analizziamo il caso del Tag helper Image, che serve per generare un'immagine HTML a cui viene aggiunto un numero di versione, così da non avere problemi con la cache lato client del browser. Nell'Esempio 6.30 possiamo trovare un esempio di come attivare il tag.

Esempio 6.30

```
<img src=″~/images/brand.png″
  asp-append-version=″true″ />
```

L'output generato sarà molto simile a quello dell'Esempio 6.30.

Esempio 6.31

```
<img src=″ /images/brand.png?v=APzJduEowspUX8NrqJLwjZrRj3SflDUbO3UjD9MFgy0″ />
```

Il valore assegnato al parametro `v` è quello dell'hash Sha512 del file su disco, che viene calcolato e tenuto in cache, per essere invalidato e calcolato di nuovo ad ogni modifica del file. Il risultato è che il browser, ricevendo un URL differente, provvederà a scaricare nuovamente il file, in caso di cambiamento.

Conclusioni

In questo capitolo abbiamo chiuso il cerchio sul pattern model-view-controller, entrando nel dettaglio sulle modalità con cui, in ASP.NET Core MVC, possiamo realizzare le view, ossia i componenti ultimi che servono a produrre l'effettivo markup HTML che verrà visualizzato sul browser utente.

Una view è un file che integra al suo interno markup e codice C# e che viene processato da un view engine che prende il nome di Razor. Si tratta, a tutti gli effetti, di un "linguaggio" inedito, per cui inizialmente ci siamo soffermati sulla sua sintassi di base.

Successivamente, abbiamo illustrato gli strumenti che abbiamo a disposizione in questa tecnologia: grazie alle layout view, siamo in grado di realizzare dei template comuni a tutte le pagine, così che il nostro sito web appaia consistente dal punto di vista grafico. Gli HTML helper, invece, offrono un supporto differente, orientato alla generazione del markup: è il caso di ActionLink e RouteLink, grazie ai quali possiamo creare link alle pagine in base alle impostazioni di routing, o di Partial che, rispettivamente, sfruttano il concetto di partial view per componentizzare porzioni di interfaccia, così che siano riutilizzabili in molteplici pagine. Inoltre, abbiamo anche visto come con i Tag helper possiamo sfruttare una nuova modalità, maggiormente orientata all'inserimento di tag HTML speciali nella pagina.

Tuttavia, a questo punto, siamo ancora a metà del nostro percorso di apprendimento. Infatti non siamo ancora in grado di gestire correttamente l'interazione utente e di accettare l'input di dati. Si tratta di un argomento piuttosto vasto, che affronteremo nel prossimo capitolo, dove daremo ancora maggior risalto, come anticipato, all'utilizzo dei Tag helper.

Definire il model

Display

Display Name

ResourceType

Name

```
//Stringa normale
[Display(Name=″Indirizzo″)]
public string Address { get; set; }

//Stringa da file di risorse
[Display(Name=″Indirizzo″, ResourceType=typeof(Labels))]
public string Address { get; set; }
```

Display

DisplayFormat

ToString

DisplayFormat

DisplayFormat

DataFormatString

ApplyFormatInEditMode

false

DisplayText NullDisplayText

DisplayFormat

```
[DisplayFormat(DataFormatString = "{0:d}",
  ApplyFormatInEditMode = True)]
public DateTime RegistrationDate { get; set; }
```

DisplayFormat

UIHint

UIHint

Il concetto di template non è stato ancora introdotto ma ne parleremo approfonditamente nel prosieguo del capitolo.

UIHint UIHint

```
[UIHint(″Address″)]
public string Address { get; set; }
```

Display DisplayFormat UIHint

```
public class CustomerModel
{
  public CustomerModel()
  {
    Countries = new List<CountryModel>();
```

```
      Contacts = new List<ContactModel>();
    }
    public int Id { get; set; }
    [Display(Name = ″Nome″)]
    public string Name { get; set; }
    [DisplayFormat(DataFormatString = ″{0:d}″,
      ApplyFormatInEditMode=true)]
    [Display(Name = ″Data registrazione″)]
    public DateTime RegistrationDate { get; set; }
    [DisplayFormat(DataFormatString = ″{0:n2}″,
      ApplyFormatInEditMode = true)]
    [Display(Name = ″Sconto″)]
    public decimal DiscountPercentage { get; set; }
    [Display(Name = ″Attivo″)]
    public bool IsActive { get; set; }
    [Display(Name = ″Indirizzo″)]
    public string Address { get; set; }
    [Display(Name = ″Stato″)]
    public int CountryId { get; set; }
    public List<CountryModel> Countries { get; set; }
  }
  public class CountryModel
  {
    public int Id { get; set; }
    public string Name { get; set; }
  }
```

Definire il controller e la action

CustomersController Create

```
  public class CustomersController : Controller
  {
    public IActionResult Create()
    {
      var model = new CustomerModel();
      model.Countries.AddRange(GetCountries()); //recupera gli stati
      return View(model);
```

```
    }
    [HttpPost]
    public IActionResult Create(CustomerModel model)
    {
      ...
    }
  }
```

HttpPost

CustomerModel

Creare la view

BeginForm

```
<form method="post"
  asp-controller="Customers"
  asp-action="Create">
  … resto della form
  <input type="submit" value="Invia dati" />
</form>
```

asp-route-*

asp-route

asp-controller asp-action

```
<form method="post" action="/Customers/Create">
  … resto della form e dei pulsanti
  <input name="__RequestVerificationToken" type="hidden" value="…rimosso…" />
</form>
```

TextBox TextBoxFor
MvcHtmlString input
type text

TextBox
TextBoxFor

Il metodo TextBoxFor offre una sintassi tipizzata, quindi è da preferire al metodo TextBox. Tuttavia, non esistono solo TextBox e TextBoxFor; tutti gli HTML helper che generano campi di input hanno una versione tipizzata e una non tipizzata (come vedremo nel corso del capitolo) e quella tipizzata è da preferire sempre.

```
@Html.TextBoxFor(m => m.Name)
@Html.TextBox("Name")
<input type="text" asp-for="Name" />
```

big

```
@Html.TextBoxFor(m => m.Name, new { @class = ″big″ })
@Html.TextBox(″Name″, new { @class = ″big″ })
<input type=″text″ asp-for=″Name″ class=″big″ />
```

asp-for

asp-for

ModelExpression

m => m.Name

ModelState

Model.Name Name

Address.Country

```
  <input type=″text″ asp-for=″IsActive″ />
```

	type
String	text
Bool	checkbox
DateTime	datetime
[DataType(DataType.Date)]	date
[DataType(DataType.Time)]	time
Byte Int Single Double	number
[HiddenInput]	hidden
[Url]	url
[DataType(DataType.Password)]	password
[Email]	email

```
<input type=″hidden″ asp-for=″Id″ />
```

[HiddenInput] Id

asp-for

asp-items

SelectListItem

```
@{
  var countries = Model.Countries.Select(c =>
    new SelectListItem() {
      Text = c.Name,
      Value = c.Id.ToString()
  });
}
<select asp-for="CountryID" asp-items="@countries">
  <option value="">Seleziona un valore...</option>
</select>
```

Nell'Esempio 7.12 abbiamo trasformato una lista di oggetti CountryModel in una lista di oggetti SelectListItem. Volendo, possiamo anche modificare il model per trasformare la proprietà Countries da List<CountryModel> a List<SelectListItem> così da non dover effettuare la trasformazione nella view. La scelta dell'uso di una tecnica piuttosto che di un'altra è strettamente personale, in quanto il risultato che si ottiene è il medesimo. Questo controllo consente anche di specificare un gruppo (attraverso la proprietà Group).

multiple="multiple"

asp-for IEnumerable

Textarea

[MinLength] [MaxLength]

```
public class Customer
{
  … altre proprietà
  [MinLength(10)]
  [MaxLength(1000)]
  public string Notes { get; set; }
```

```
<textarea asp-for=″Notes″></textarea>
```

Lo stesso effetto può essere ottenuto utilizzando la data annotation [StringLength], nella forma [StringLength(maximumLength: 1000, MinimumLength = 10)].

label

Display

Display

```
<label asp-for="Name"></label>
```

Editor EditorFor

Editor EditorFor

I template di default sono contenuti all'interno degli assembly di ASP.NET Core MVC. Tuttavia possiamo personalizzare i template come viene mostrato nel prosieguo di questa sezione.

Editor EditorFor

```
@Html.EditorFor(m => m.Name)
@Html.Editor("Name")
```

nometipo.cshtml Views/Shared/EditorTemplates

DateTime.cshtml

```
@model DateTime
@Html.TextBox(″″, Model, new { @class = ″date″ })
```

TextBox

TextBoxFor

DateTime

UIHint

UIHint

Editor EditorFor

Display DisplayFor

DisplayFormat

Display DisplayFor

```
@Html.DisplayFor(m => m.Name)
@Html.Display("Name")
```

Editor EditorFor Display DisplayFor

MyFirstApp Home About Contact

Customer

Daniele Daniele

© 2018 - MyFirstApp

CustomerModel

CustomerModel

jquery.validate

Gli attributi di validazione

Required
Range RegularExpression StringLength Remote

ValidationAttribute

- ErrorMessage
- ErrorMessageResourceType
- ErrorMessageResourceName

ErrorMessageResourceType

Required

Required

string

Int32 Decimal Boolean

Required Name

```
[Required]
public string Name { get; set; }
```

Range

DiscountPercentage

Range

Range MinimumValue

MaximumValue

```
[Range(0, 100)]
public decimal DiscountPercentage { get; set; }
```

RegularExpression

Name

Pattern

```
[RegularExpression(″[A-Za-z ´àéèìòù]+″)]
public string Name { get; set; }
```

StringLength

StringLength

MinimumLength MaximumLength

```
[StringLength(40, MinimumLength = 1)]
public string Name { get; set; }
```

Remote

Bool

true false

```
public class Customer
{
  // … altre proprietà
  [Remote(action: ″IsNameValid″, controller: ″Customers″)]
  public string Name { get; set; }
}
```

```
[AcceptVerbs(″Get″, ″Post″)]
public IActionResult IsNameValid(string Name)
{
  var isValid = false;
  //Logica di validazione
  return Json(isValid);
}
```

- CreditCard
- Compare
- EmailAddress
- Phone
- Url

Applicare la validazione sulla view

jquery.validate

data-

data-

Scripts

_ValidationScriptsPartial.cshtml

```
<environment include="Development">
    <script src="~/lib/jquery-validation/dist/jquery.validate.js"></script>
    <script src="~/lib/jquery-validation-unobtrusive/jquery.validate.unobtrusive.js">
</script>
</environment>
<environment exclude="Development">
    <script src="https://ajax.aspnetcdn.com/ajax/jquery.validate/1.17.0/
    jquery.validate.min.js"
            asp-fallback-src="~/lib/jquery-validation/dist/jquery.validate.min.js"
            asp-fallback-test="window.jQuery && window.jQuery.validator"
            crossorigin="anonymous"
            integrity="sha384-rZfj/ogBloos6wzLGpPkkOr/gpkBNLZ6b6yLy4o+ok+t/
            SAKlL5mvXLr0OXNi1Hp">
    </script>
    <script src="https://ajax.aspnetcdn.com/ajax/jquery.validation.unobtrusive/
            3.2.9/jquery.validate.unobtrusive.min.js"
            asp-fallback-src="~/lib/jquery-validation-unobtrusive/jquery.validate.
            unobtrusive.min.js"
            asp-fallback-test="window.jQuery && window.jQuery.validator &&
            window.jQuery.validator.unobtrusive"
            crossorigin="anonymous"
            integrity="sha384-ifv0TYDWxBHzvAk2Z0n8R434FL1Rlv/Av18DXE43N/1rvHyOG4iz
Kst0f2iSLdds">
    </script>
</environment>
```

_ValidationScriptsPartial.cshtml

Shared

ValidationMessage ValidationMessageFor

span asp-validation-for

```
@Html.LabelFor(m => m.Name)
@Html.EditorFor(m => m.Name)
@Html.ValidationMessageFor(m => m.Name)
```

```
<div class="form-group">
    <label asp-for="Name" class="col-md-2 control-label"></label>
    <div class="col-md-10">
        <input asp-for="Name" class="form-control" />
        <span asp-validation-for="Name" class="text-danger"></span>
    </div>
</div>
```

asp-validation-summary div

```
<div asp-validation-summary="ModelOnly"></div>
```

Personalizzare la validazione

ValidationAttribute

IsValid

ValidationContext

ObjectInstance

ValidationResult.Success

ValidationResult

Il primo overload a essere eseguito da ASP.NET Core MVC è quello che accetta il contesto. Se il metodo invoca la versione base, successivamente viene invocato il metodo che accetta solo il valore da validare. Se invece il metodo che accetta il contesto restituisce un risultato, la validazione termina e il metodo che accetta solo il valore non viene invocato.

```
public class CodiceFiscaleAttribute : ValidationAttribute
{
  public override bool IsValid(object value)
  {
    var result = false;
    //valida codice fiscale
    return result;
  }
  protected override ValidationResult IsValid(object value,
    ValidationContext validationContext)
  {
    var result = false;
    //valida codice fiscale
    if (result)
      return ValidationResult.Success;
    else
      return new ValidationResult(″Codice fiscale errato″);
  }
}
```

jquery.validate

ModelClientCodiceFiscaleValidationRule
IClientModelValidator
AddValidation

CodiceFiscale
IClientValidatable
GetClientValidationRules

ModelClientCodiceFiscaleValidationRule

```
public class CodiceFiscaleAttribute : IClientModelValidator
{
  ...
  public void AddValidation(ClientModelValidationContext context)
  {
    if (context == null)
    {
      throw new ArgumentNullException(nameof(context));
    }
    MergeAttribute(context.Attributes, ″data-val″, ″true″);
    MergeAttribute(context.Attributes, ″data-val-codicefiscale″,
GetErrorMessage());
    MergeAttribute(context.Attributes, ″data-val-codicefiscale-value″, value);
  }
}
```

jquery.validate
codicefiscale

```
jQuery.validator.addMethod(″codicefiscale″,
function (value, element, param) {
  var isValid = false;
  //valida codice fiscale
  return isValid;
});

jQuery.validator.unobtrusive.adapters.add(″codicefiscale″, [],
  function (options) {
    options.rules[″codicefiscale″] = options.params;
    options.messages[″codicefiscale″] = options.message;
});
```

codicefiscale
jquery.validate

IsValid

IValidatableObject
Validate

ValidationResult

IValidatableObject

```
public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
{
  var minimumYear = DateTime.Today.AddYears(-5).Year;
    if (!IsActive && RegistrationDate.Year < minimumYear)
    {
         yield return new ValidationResult(
              $"Gli utenti attivi devono essere registrati prima del
{minimumYear}.",
              new[] { "RegistrationDate" });
    }
}
```

Gestire gli errori nella action

ModelState.IsValid
true ModelState.Values

ModelState.Values
Value
Errors

```
[HttpPost]
public IActionResult Create(CustomerModel model)
{
  if (ModelState.IsValid)
  {
    //Salva dati
    return RedirectToAction(″Conferma″);
  }
  model.Countries.AddRange(GetCountries());
  return View(model);
}
```

ModelState

id

id

CustomerModel

Name

Address

Address City ZipCode

Address.Address

Address.City

Address.ZipCode

name

8

La gestione dello stato

stateless

Come funziona una richiesta HTTP?

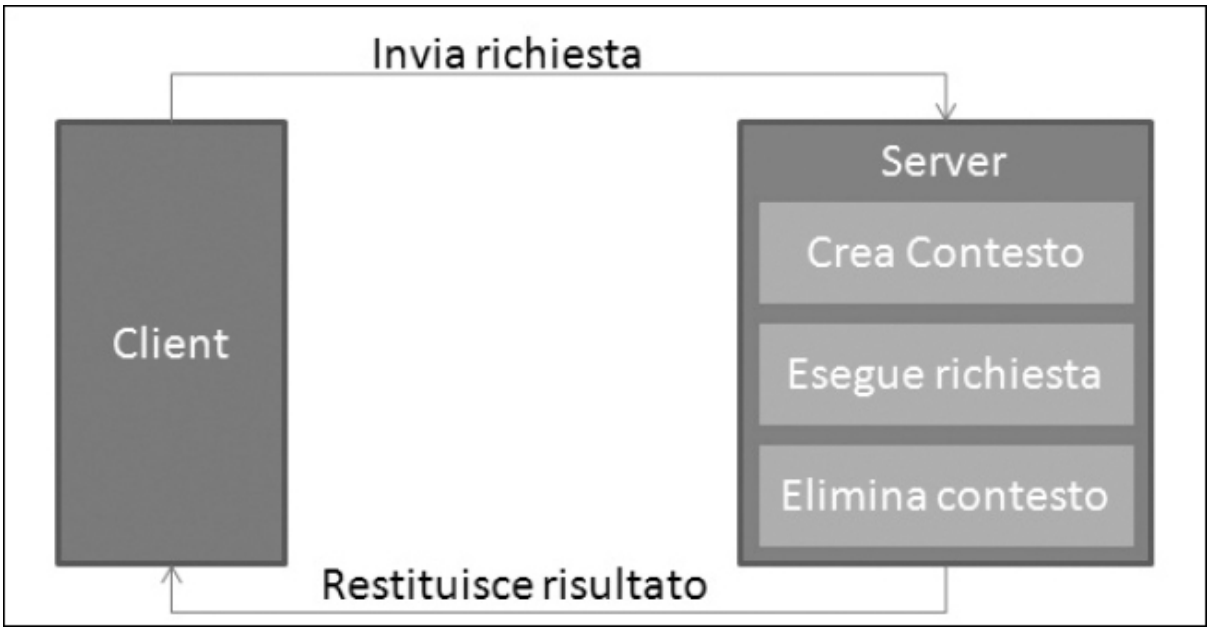


Figura 8.1 – Il flow di una richiesta web.

Scenari di gestione dello stato

- ❑

-
-
-

Lo stato con i campi hidden

Esempio 8.1

```
<input type=″hidden″ asp-for=″Id″ />
```

Esempio 8.2

```
document.querySelector(´#@Html.IdFor(m => m.Id)´).value = ´new value´;
```

Lo stato attraverso i cookie

cookie

Un cookie può anche essere generato sul client tramite JavaScript. Una volta generato, questo cookie segue la stessa strada di tutti gli altri cookie. Questa tecnica è poco usata in quanto molte applicazioni bloccano questo tipo di cookie.

Response IHttpContextAccessor Response Cookies IResponseCookies

- Append

- Delete

Append

Delete

Esempio 8.3

```
Response.Cookies.Append(key, value);
Response.Cookies.Delete(key);
```

Append

CookieOptions

Tabella 8.1 – Le proprietà della classe CookieOptions

Attributo	Descrizione
Domain	
Expires	
HttpOnly	
IsEssential	
MaxAge	max-age
Path	

SameSite	http://aspit.co/ai9
Secure	

Request
IHttpContextAccessor

Cookies IRequestCookieCollection

- Keys
-
- TryGetValue

IRequestCookieCollection IEnumerable<KeyValuePair>

Esempio 8.4

```
public IActionResult ReadCookies()
{
  var model = new Dictionary<string, string>();
  foreach(var item in Request.Cookies)
  {
    model.Add(item.Key, item.Value);
  }
```

```
    var value = Request.Cookies[″CookieKey″];
    Request.Cookies.TryGetValue(″CookieKey″, out var tryvalue);
    return View(model);
  }
```

foreach

TryGetValue

Gestione dello stato nella sessione

cookie di sessione

SessionMiddleware

UseSession Configure

Startup UseSession

SessionOptions

IOTimeout

IdleTimeout

AddSession
ConfigureServices Startup AddSession
DistributedSessionStore
IDistributedCache

AddDistributedMemoryCache

AddSession
UseSession AddSession

SessionOptions

Esempio 8.5

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddDistributedMemoryCache();
  //senza parametri
  services.AddSession();
  //con configurazione delle opzioni
  services.AddSession(a => a.IdleTimeout = TimeSpan.FromMinutes(5));
}
public void Configure(IApplicationBuilder app,
  IHostingEnvironment env)
{
  //senza parametri
  app.UseSession();
  //con configurazione delle opzioni
  app.UseSession(new SessionOptions {
    IdleTimeout = TimeSpan.FromMinutes(5))
  }
}
```

HttpContext

IHttpContextAccessor
Session ISession
TryGetValue
Set
Remove Clear

IsAvailable

TryGetValue Set

GetString GetInt32 SetString SetInt32

Esempio 8.6

```
public IActionResult Session()
{
  HttpContext.Session.SetString("key", "value");
  var value = HttpContext.Session.GetString("key");
  HttpContext.Session.Remove("key");
}
```

SetString
GetString

Esempio 8.7

```
public static class SessionExtensions
{
  public static void SetValue(this ISession session, string key,
```

```
      object value)
    {
      session.SetString(key, JsonConvert.SerializeObject(value));
    }
    public static T GetValue<T>(this ISession session, string key)
    {
      var value = session.GetString(key);
      return value == null ?
        default(T) :
        JsonConvert.DeserializeObject<T>(value);
    }
  }
```

Come abbiamo detto in precedenza, la sessione si basa sullo store della cache che implementa l'interfaccia IDistributedCache. Il metodo AddDistributedMemoryCache aggiunge uno store che salva la sessione in memoria, ma questo non è l'unico metodo disponibile. Esistono i metodi AddDistributedSqlServerCache e AddDistributedRedisCache che configurano rispettivamente Sql Server e Redis come store. Questi metodi verranno analizzati più avanti nel capitolo, quando parleremo della cache. Per ora basti sapere che qualunque provider utilizziamo per la cache, questo verrà utilizzato anche per la sessione.

Passare valori tramite querystring

?

{chiave}=
{valore} &
http://www.sito.it/home/qs?
key=1&v=A
key v
1 A

Url.Action
a
Html.ActionLink a

Request HttpRequest

Request HttpContext
IHttpContextAccessor HttpRequest
QueryString,
Query

Esempio 8.8

```
public IActionResult QS(int key, string v)
{
  var model = new QSModel();
  model.KeyFromParam = key;
  model.VFromParam = v;
  model.KeyFromQS = Request.Query["key"];
  model.VFromQS = Request.Query["v"];
}
```

Request

Non esiste un limite per la lunghezza di un url, quindi in teoria possiamo mettere in querystring quanti dati vogliamo. Tuttavia non essendoci specifiche, ogni browser può imporre un proprio limite e anche i web server possono avere un loro limite di lunghezza. Per questi motivi è meglio evitare di mettere troppi dati in querystring (1-2 KB, per dare un'idea di massima).

Gestione dei dati temporanei con TempData

AddSessionStateTempDataProvider

ConfigureServices

Esempio 8.9

```
public void ConfigureServices(IServiceCollection services)
{
  services
    .AddMvc()
    .AddSessionStateTempDataProvider();
  services.AddSession();
}
```

AddSession

AddSessionStateTempDataProvider

ITempDictionary

Add

Peek

Esempio 8.10

```
public IActionResult TempDataAdd()
{
  TempData.Add(″Key″, ″value″);
  return Content(String.Empty);
}
public IActionResult TempDataPeek()
{
  var value = TempData.Peek(″Key″);
  return Content((string)value);
}
```

```
  public IActionResult TempDataGet()
  {
    var value = TempData["Key"];
    return Content((string)value);
  }
```

TempDataAdd

TempDataPeek

Peek

TempDataGet

Mantenere dati di una richiesta: HttpContext.Items

HttpContext.Items

IDictionary

HttpContext.Items

Esempio 8.11

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
  // Aggiunge un middleware che crea il timer
  app.Use(async (context, next) =>
  {
    var sw = new Stopwatch();
    context.Items["sw"] = sw;
    sw.Start();
    await next.Invoke();
    sw.Stop();
    Debug.WriteLine(sw.ElapsedMilliseconds);
  });
}

// Crea una action che legge il timer
public IActionResult HttpItems()
{
  var sw = (Stopwatch)HttpContext.Items["sw"];
  return Content(sw.ElapsedMilliseconds.ToString());
}
```

HttpContext

IHttpContextAccessor

Utilizzare la cache

IMemoryCache **MemoryCache**

IDistributedMemoryCache

- **SqlServerCache**
- **RedisCache**
- **MemoryDistributedCache**

Local cache

AddMemoryCache
ConfigureServices Startup
AddMemoryCache

MemoryCacheOptions
SizeLimit,

ExpirationScanFrequency

Esempio 8.12

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddMemoryCache(o => {
    // Imposta opzioni cache
  });
}
```

IMemoryCache

CreateEntry
Remove TryGetValue

CreateEntry TryGetValue

Get Set
GetOrCreateAsync Remove
Set

DateTimeOffset
TimeSpan

MemoryCacheEntryOptions

Esempio 8.13

```
public IActionResult Cache([FromServices] IMemoryCache cache)
{
  // elemento senza opzioni
  cache.Set(″key″, ″value″);

  // elemento che scade dopo un´ora
  cache.Set(″key″, ″value″, DateTime.Now.AddHours(1));

  // elemento che scade dopo 10 minuti dall´ultimo accesso
  cache.Set(″key″, ″value″, TimeSpan.FromMinutes(10));

  // elemento che scade dopo 10 minuti dall´ultimo accesso
  cache.Set(″key″, ″value″, new MemoryCacheEntryOptions {
    SlidingExpiration = TimeSpan.FromMinutes(10) });
}
```

Get
object

Esempio 8.14

```
object value = cache.Get(″key″);
string stringValue = cache.Get<string>(″key″);
```

GetOrCreate

GetOrCreateAsync

Key

value

Esempio 8.15

```
var value = cache.GetOrCreate(″key″, e =>
{
  e.AbsoluteExpiration = DateTime.Now.AddMinutes(10);
  return ″value″;
});
var valueAsync = await cache.GetOrCreateAsync(″key″, e =>
{
  e.AbsoluteExpiration = DateTime.Now.AddMinutes(10);
  return SomeAsyncMethod();
});
```

Remove

Esempio 8.16

```
cache.Remove(″key″);
```

Gestione avanzata della local cache

MemoryCacheEntryOptions

AbsoluteExpiration SetAbsoluteExpiration

SlidingExpiration SetSlidingExpiration

Priority

CacheItemPriority

SetPriority

PostEvictionCallbacks

RegisterPostEvictionCallback

IChangeToken

PollingFileChangeToken

CancellationChangeToken

CancellationTokenSource

Cancel

Esempio 8.17

```
public IActionResult Cache()
{
  var cts = new CancellationTokenSource();
  cache.Set(″keyDep″, cts);

  var value = cache.GetOrCreate(″key″, e =>
  {
    e.SetAbsoluteExpiration(DateTime.Now.AddMinutes(10));
    e.SetSlidingExpiration(TimeSpan.FromMinutes(1));
    e.ExpirationTokens.Add(new CancellationChangeToken(cts.Token));
    e.ExpirationTokens.Add(new PollingFileChangeToken(new
      FileInfo(@″c:\file.txt″)));
    e.RegisterPostEvictionCallback(OnEviction);
    return ″value″;
  });
}

public IActionResult CacheExpire()
{
  cache.Get<CancellationTokenSource>(″keyDep″).Cancel();
  return Content(″cancelled″);
}

private void OnEviction(object key, object value,
  EvictionReason reason, object state)
{
  Debug.WriteLine($″{key} removed. Reason:{reason}″);
}
```

CacheExpire

Distributed cache

AddDistributedMemoryCache
ConfigureServices Startup

MemoryDistributedCacheOptions
SizeLimit,
ExpirationScanFrequency

Esempio 8.18

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddDistributedMemoryCache(o => {
    // Imposta opzioni cache
  });
}
```

AddDistributedSqlServerCache

SqlServerCacheOptions
ConnectionString
TableName SchemaName

Esempio 8.19

```
services.AddDistributedSqlServerCache(o =>
{
```

```
    o.ConnectionString = ″connection string″;
    o.TableName = ″DistributedCache″;
    o.SchemaName = ″dbo″;
  });
```

Esempio 8.20

```
  dotnet sql-cache create ″connection string″ dbo DistributedCache
```

Configure Startup
AddDistributedRedisCache

RedisCacheOptions
Configuration
InstanceName

Esempio 8.21

```
  services.AddDistributedRedisCache(o =>
  {
    o.Configuration = ″connection string″;
    o.InstanceName = ″db1″;
  });
```

IDistributedMemoryCache

Get Set Remove
SetString GetString

Set SetAsync

SetString SetStringAsync

Esempio 8.22

```
public async Task<IActionResult> Cache(
  [FromServices] IDistributedCache cache)
{
  // elemento senza opzioni
  await cache.SetAsync(″key″, Serialize(″value″));

  // elemento che scade dopo un´ora
  await cache.SetAsync(″key″, Serialize(″value″),
    new DistributedCacheEntryOptions
    {
      AbsoluteExpiration = DateTime.Now.AddHours(1)
    });

  // elemento che scade dopo 10 minuti dall´ultimo accesso
  await cache.SetAsync(″key″, Serialize(″value″),
    new DistributedCacheEntryOptions
    {
      SlidingExpiration = TimeSpan.FromMinutes(10)
    });

  // elemento inserito usando SetString
  await cache.SetStringAsync(″key″, ″value″);

  // elemento con un oggetto inserito usando SetString
  await cache.SetStringAsync(″key″, SerializeJson(obj));
```

```
}
```

Get GetAsync

GetString/ GetStringAsync
Get GetAsync

Esempio 8.23

```
// recupera valore come array di byte
byte[] value = await cache.GetAsync("key");

// recupera valore come stringa
string value = await cache.GetStringAsync("key");

// recupera valore serializzato in json e lo deserializza in un tipo
var value = DeserializeJson<MyClass>(
  await cache.GetStringAsync("key"));
```

Remove RemoveAsync

Esempio 8.24

```
await cache.RemoveAsync("key");
```

HTML caching

cache distributed-cache

cache distributed-cache,

Esempio 8.25

```
<distributed-cache name=″cache-key-1″>
  Time: @DateTime.Now
</distributed-cache>

<cache>
  Time: @DateTime.Now
<cache>
```

Tabella 8.2 – Gli attributi dei tag helper di cache.

Attributo	Descrizione

enabled	
expires-on	
expires-after	
expires-sliding	
vary-by-header	
vary-by-query	
vary-by-route	
vary-by-cookie	
vary-by-user	
vary-by	
priority	
name	distributed-cache

Esempio 8.26

```
<cache
  expires-on="@new DateTime(2018, 5, 6, 18, 0, 0)"
  expires-sliding="@TimeSpan.FromSeconds(60)"
  vary-by-user="true"
  vary-by-header="accept-language">
    Content: @DateTime.Now - @User.Identity.Name
</cache>
```

expires-on expires-sliding

accept-language vary-by-user vary-by-header

Response caching

UseResponseCaching AddResponseCaching

ResponseCacheAttribute

ResponseCacheAttribute
Duration

Location
ResponseCacheLocation
None
Client Any

VaryByHeader VaryByQueryKeys

NoStore
true

ResponseCacheAttribute

Esempio 8.27

```
// Mette in cache per 10 secondi
[ResponseCache(Duration = 10)]

// Mette in cache per 10 secondi creando una copia per ogni valore del header
Accept-Lang
[ResponseCache(Duration = 10, VaryByHeader = ″Accept-Language″)]

// Mette in cache solo sul client per 5 secondi
[ResponseCache(Duration = 5, Location = ResponseCacheLocation.Client)]

// Disabilita la cache
[ResponseCache(NoStore = true)]
```

CacheProfileName

ResponseCacheAttribute

Esempio 8.28

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddMvc(o =>
  {
    o.CacheProfiles.Add(″Default″,
     new CacheProfile() { Duration = 60 });
  });
}

[ResponseCache(CacheProfileName = ″Default″)]
public IActionResult CacheableAction(string id)
{
  ...
}
```

Conclusioni

- System.Data.Common
- System.Data. Common

System. Data.SqlClient

- System.Data.SqlTypes
- System.Data

Il namespace System.Data.Common

System.Data.Common

- DbConnection
- DbCommand
- DbDataReader

DbCommand

- DbTransaction
- DbParameter
- DbConnectionStringBuilder
- DbDataAdapter

System.Data

Data provider

System.Data. SqlClient

- SqlConnection DbConnection
- SqlCommand DbCommand
- SqlDataReader DbDataReader

  SqlCommand
- SqlTransaction DbTransaction
- SqlParameter DbParameter
- SqlConnectionStringBuilder
  DbConnectionStringBuilder

- `SqlDataAdapter` `DbDataAdapter`

http://aspit.co/r9

Oledb e Odbc sono stati portati in .NET Core 2.0 ma non fanno parte di .NET Standard 2.0, quindi possiamo utilizzarli solo se stiamo creando un'applicazione basata sul primo.

Il namespace System.Data.SqlTypes

`System.Data.SqlTypes`

System.Data.SqlTypes
SqlString SqlInt32 SqlByte SqlBinary SqlBoolean SqlDateTime

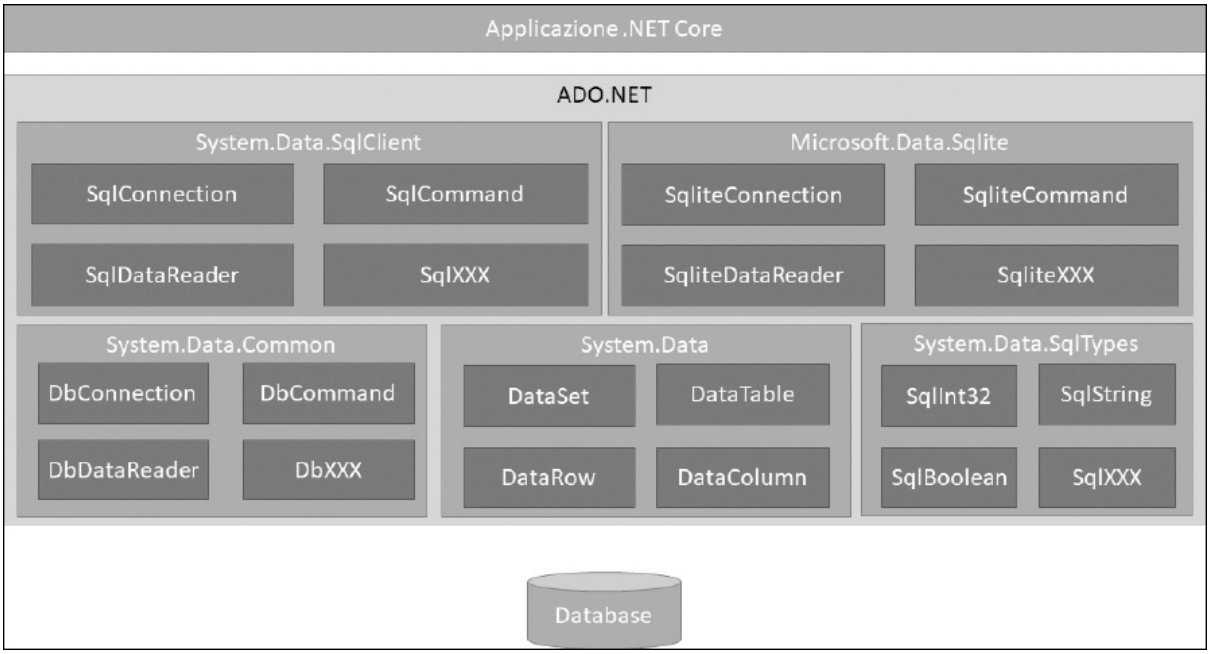
Il namespace System.Data

System.Data

- DataTable
- DataRow
  DataTable
  DataRow
- DataColumn
  DataRow
  DataColumn
- DataSet
  DataTable

DataSet DataTable
DataRow

DataColumn



Stabilire una connessione

DbConnection
ConnectionString

Open OpenAsync Close

SqlConnection

```
// Stringa di connessione
var connectionString = ″Server=localhost;Database=Northwind;
  User ID=appUser; Password=p@$$w0rd″;

// Creazione dell´istanza di SqlConnection
var conn = new SqlConnection(connectionString);

try
{
  // Apertura della connessione
  await conn.OpenAsync();
  // ...
}
catch(SqlException ex)
{
  // Gestione dell´eccezione
}
finally
{
  // Chiusura della connessione
  if (conn.State == ConnectionState.Open)
```

```
    conn.Close();
}
```

DbConnection IDisposable using

```
var connectionString = ″...″;
using(var connection = new SqlConnection(connectionString))
{
  try
  {
    await connection.OpenAsync();
    // ...
  }
  catch(SqlException ex)
  {
    // Gestione dell´eccezione
  }
}
```

using Dispose

- Data Source Server
- Database Initial Catalog
- User ID Uid

- Password Pwd

- Integrated Security Trusted_Connection

ConnectionStrings

```
{
  "ConnectionStrings": {
    "SQL": "Server=dbserver;Database=northwind;Integrated security=true"
  }
}
```

GetConnectionString IConfiguration

IConfiguration
GetConnectionString

```
public class CustomerService
{
  private readonly string connectionString;
  public CustomerService(IConfiguration configuration)
  {
    connectionString = configuration.GetConnectionString(″SQL″);
  }
}
```

DbConnectionStringBuilder
DbConnectionStringBuilder

ConnectionString

SqlConnectionStringBuilder

```
var builder = new SqlConnectionStringBuilder();
builder.DataSource = serverName;
builder.InitialCatalog = database;
builder.IntegratedSecurity = true;
var connectionString = builder.ConnectionString;
```

Esecuzione di un comando

DbCommand

CommandText	
CommandType	CommandType Text StoredProcedure TableDirect
Connection	
Parameters	
Transaction	
ExecuteNonQuery ExecuteNonQueryAsync	

	Task<int>
ExecuteReader ExecuteReaderAsync	Task<DbDatareader>
ExecuteScalar ExecuteScalarAsync	object Task<object>

ExecuteNonQuery ExecuteNonQueryAsync

ExecuteReader ExecuteReaderAsync

ExecuteScalar ExecuteScalarAsync

SELECT COUNT SELECT MAX SELECT MIN

```
//Metodo sincrono
private void Method(SqlConnection connection)
{
  // Aggiornamento
  var cmdUpdate = new SqlCommand(″UPDATE Products SET ...″, connection);
  int affectedRows = cmdUpdate.ExecuteNonQuery();

  // Query
  var cmdQuery = new SqlCommand(″SELECT * FROM Products″, connection);
  SqlDataReader reader = cmdQuery.ExecuteReader();

  // Conteggio
  var cmdCount = new SqlCommand(″SELECT COUNT(*) FROM Products″, connection);
  var count = (int)cmdCount.ExecuteScalar();
}

//Metodo asincrono
private async Task MethodAsync(SqlConnection connection)
{
  // Aggiornamento
  var cmdUpdate = new SqlCommand(″UPDATE Products SET ...″, connection);
  int affectedRows = await cmdUpdate.ExecuteNonQueryAsync();

  // Query
  var cmdQuery = new SqlCommand(″SELECT * FROM Products, connection ″);
  SqlDataReader reader = await cmdQuery.ExecuteReaderAsync();

  // Conteggio
  var cmdCount = new SqlCommand(″SELECT COUNT(*) FROM Products″, connection);
  var count = (int)(await cmdCount.ExecuteScalarAsync());
}
```

DbCommand

CommandType

System.Data.CommandType

StoredProcedure

CommandText

DbParameter

Add Parameters

```
var query = new SqlCommand(
  ″SELECT * FROM Orders ″ +
  ″WHERE EmployeeID = @EmployeeID ″ +
  ″AND OrderDate = @OrderDate ″ +
  ″AND ShipCountry = @ShipCountry″ +
  ″ORDER BY OrderDate DESC″, connection);
var p1 = new SqlParameter
{
  ParameterName = ″@EmployeeID″;
  DbType = DbType.Int32;
  Direction = ParameterDirection.Input;
  Value = 1;
}
var p2 = new SqlParameter {
  ParameterName = ″@OrderDate″;
  DbType = DbType.DateTime;
  Direction = ParameterDirection.Input;
  Value = new DateTime(1996, 8, 7);
}
var p3 = new SqlParameter
{
  ParameterName = ″@ShipCountry″;
  DbType = DbType.String;
  Direction = ParameterDirection.Input;
  Value = ″Italy″;
}
query.Parameters.Add(p1);
query.Parameters.Add(p2);
query.Parameters.Add(p3);
```

Scrivere dati in transazione

DbTransaction
BeginTransaction
DbConnection BeginTransaction
DbTransaction Commit
Rollback

```
using (var connection = new SqlConnection(connectionString))
{
  connection.Open();
  using (SqlTransaction transaction = connection.BeginTransaction())
  {
    try
    {
      // inserisce un ordine
      // inserisce i dettagli
      // aggiorna il magazzino
      transaction.Commit();
    }
    catch (SqlException ex)
    {
      transaction.Rollback();
      // Gestione dell´eccezione
    }
  }
}
```

catch

using

System.Transaction

TransactionScope

async/await

```
public void CreateOrder(Order order)
{
  using (var scope = new TransactionScope())
  {
    SaveOrder(order);
    UpdateStock(order);
    scope.Complete();
  }
}
private SaveOrder(Order order)
{
  using (var connection = new SqlConnection(connectionString))
  {
    connection.Open();
    //comandi di salvataggio dell´ordine
  }
}
private UpdateStock(Order order)
{
  using (var connection = new SqlConnection(connectionString))
  {
    connection.Open();
    //comandi di aggiornamento del magazzino prodotti
  }
}
```

TransactionScope

Complete TransactionScope

using

Complete

DbTransaction

Lettura del risultato di una query

ExecuteReader
ExecuteReaderAsync

DbDataReader
System.Data.Common

Read
ReadAsync
true

❑

- GetXXX,

GetInt32 GetDateTime GetString

```
using (var cmd = new SqlCommand("SELECT * FROM Products"), connection)
{
  using (SqlDataReader reader = await cmd.ExecuteReaderAsync())
  {
    while(reader.Read())
    {
      // Viene utilizzata la proprietà indexer
      var productID = (int)reader["ProductID"];
      // ProductName è il secondo campo del record
      var productName = reader.GetString(1);
      // ...
    }
  }
}
```

Close DbDataReader IDisposable using Dispose Close

NextResult NextResultAsync *false*

```
var cmd = new SqlCommand(″SELECT * FROM Products; SELECT * FROM Customers″,
  connection);
using (SqlDataReader reader = await cmd.ExecuteReaderAsync())
{
  IterateOverProducts(reader);
  await reader.NextResultAsync();
  IterateOverCustomers(reader);
}
```

Modalità disconnessa in ADO.NET

`DataSet` `DataTable`

Le classi container di ADO.NET non sono elementi specifici di un particolare data provider. Sono altresì dei semplici contenitori, che presentano una serie di funzionalità simili a quelle offerte da un database classico, come l'organizzazione dei dati in tabelle, le relazioni, l'integrità referenziale, i vincoli, l'indicizzazione e così via. Lo scopo dei container è quello di ospitare insiemi di dati, strutturati secondo uno schema specifico, mantenendoli attivi in memoria affinché possano essere letti e modificati in modo semplice e immediato. In particolare, il `DataSet` *è un oggetto composto da un insieme di tabelle, rappresentate da altrettante istanze della classe* `DataTable` *e da relazioni di tipo* `DataRelation` *Ciascuna tabella, a sua volta, è composta da righe (classe* `DataRow`*) e colonne (classe* `DataColumn`*) e può includere vincoli di integrità referenziale e di univocità dei dati.*

`DbDataAdapter`

SelectCommand	DataSet DataTable
InsertCommand	
UpdateCommand	
DeleteCommand	
Fill(DataSet)	DataSet
Fill(DataTable)	DataTable
Update(DataSet)	DataSet
Update(DataTable)	DataTable

```
var conn = new SqlConnection(″...″);
// In fase di creazione occorre specificare la connessione
var adapter = new SqlDataAdapter(″SELECT * FROM Products″, conn);
// Creazione della DataTable
var dt = new DataTable();
// Popolamento della DataTable
adapter.Fill(dt);
// Modifica dei dati contenuti nella DataTable...
// Batch update
adapter.Update(dt);
```

Fill

Fill

Ricercare dati in un DataSet

DataTable

AsEnumerable

Where

Field

```
var ds = // popola dataset
EnumerableRowCollection<DataRow> enumDt = c.Tables[0].AsEnumerable();
var custs = enumDt.Where(t => t.Field<string>("CustomerID").StartsWith("A");
```

Where Select OrderBy

Where

10

Accedere ai dati con Entity Framework Core

Nel capitolo precedente abbiamo illustrato come ADO.NET fornisca un'ottima base per accedere ai dati. Oggetti come `DbConnection`, `DbCommand`, `DbDataReader` e `DataSet` offrono tutto ciò di cui abbiamo bisogno per interagire con un database.

Tuttavia, lavorare con questi oggetti in maniera diretta nel nostro codice significa legarlo al database e alla sua struttura. Per questo motivo, le applicazioni moderne sfruttano una tecnica più elaborata per manipolare i dati. I dati vengono recuperati dal database utilizzando le classi di ADO.NET (in uno strato dedicato all'accesso ai dati) e riversati in un insieme di classi (l'insieme è noto come object model, mentre le classi sono note come entity), che vengono restituite all'applicazione. L'applicazione manipola i dati contenuti nelle classi dell'object model e demanda allo strato di accesso ai dati la loro persistenza sul database. In questo modo, la nostra applicazione lavora principalmente con l'object model senza preoccuparsi della struttura del database se non in uno strato dedicato. Grazie all'astrazione creata dall'object model e dallo strato di accesso ai dati, la nostra applicazione è agnostica rispetto al database e quindi più semplice sia da sviluppare sia da manutenere.

Quando sviluppiamo applicazioni seguendo questo pattern, possiamo trovare un valido aiuto nei cosiddetti **Object/Relational Mapper** o **O/RM** (O/RM d'ora in poi). Un O/RM è un framework che permette di dialogare con il database astraendo le classi di ADO.NET e restituendo direttamente istanze di oggetti dell'object model. Permette anche di tracciare le modifiche fatte agli oggetti e di persisterle sul database.

In questo capitolo parleremo dell'O/RM prodotto da Microsoft: Entity Framework Core. Questo framework condivide molte cose con la sua controparte .NET (Entity Framework 6) ma, al momento della stesura di

questo libro, non espone diverse funzionalità che lo rendono meno flessibile. Tuttavia, i miglioramenti a Entity Framework Core sono continui e ben presto sarà al livello di Entity Framework 6 e lo supererà. Ne sono un esempio i miglioramenti apportati a Entity Framework Core 2.1. Questa versione, infatti, colma molte delle lacune presenti nelle precedenti versioni e rende Entity Framework Core una valida scelta.

Cosa è un O/RM

Prima di iniziare la spiegazione di Entity Framework Core, è bene accennare cosa sia un O/RM e quale idea risieda alla base di questo strumento di sviluppo. Come detto poco sopra, mascherare la struttura e l'interazione con il database dietro alle classi, permette di costruire applicazioni fortemente disaccoppiate dal database; questa è un'ottima cosa a livello di semplicità di sviluppo e manutenzione. Attraverso l'uso di un O/RM possiamo creare delle classi che rappresentino il dominio della nostra applicazione, indipendentemente da come i dati sono strutturati nel database. Sarà poi compito di uno specifico strato dell'applicazione tradurre i risultati delle query in oggetti e, viceversa, tradurre gli oggetti in comandi per aggiornare il database..

Prendendo come esempio il database `Northwind`, possiamo creare le classi `Order`, `OrderDetail` e `Customer`. In questo caso, le classi hanno una struttura speculare con le tabelle del database, ma non sempre è così. La teoria che sta dietro agli oggetti è completamente diversa dalla teoria che è alla base dei dati relazionali e questa diversità porta spesso (ma non sempre) ad avere classi diverse dalle tabelle. Il primo esempio di questa diversità risiede nella diversa **granularità**. Un cliente, in genere, ha un indirizzo di fatturazione e uno di spedizione (che possono coincidere o meno). Per rappresentare questi dati nel database, creiamo una tabella `Customers` con i campi indirizzo, C.a.p., città e nazione ripetuti per entrambe le tipologie di indirizzo. Quando invece creiamo le classi, la cosa migliore è crearne una (`AddressInfo`) con le proprietà di un indirizzo e poi nella classe `Customer` aggiungere due proprietà (`BillingAddress` e `ShippingAddress`) di tipo `AddressInfo`. Questo significa avere una tabella lato database e due classi lato Object Model, come è mostrato nella Figura 10.1.

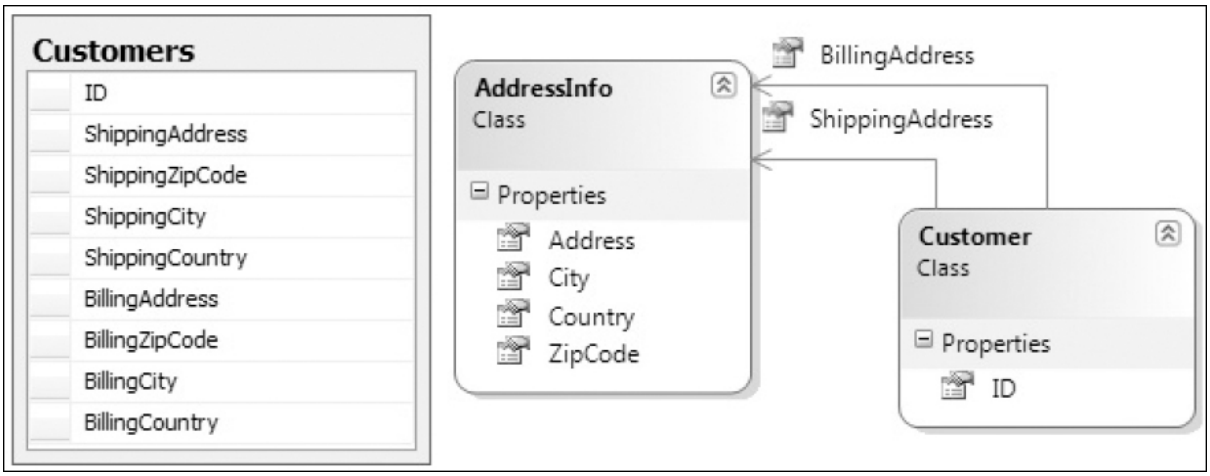


Figura 10.1 – La tabella clienti è descritta in due classi.

Un altro esempio è fornito dalla diversa modalità di relazione tra i dati. In un database, le relazioni tra i record sono mantenute tramite colonne con un vincolo di foreign key. Per esempio, per associare l'ordine a un cliente, mettiamo nella tabella degli ordini una colonna che contenga l'id del cliente. Nell'object model le relazioni si esprimono usando direttamente gli oggetti. Quindi, per mantenere l'associazione tra l'ordine e il cliente, aggiungiamo la proprietà `Customer` alla classe `Order`, come è mostrato nella figura 10.2.

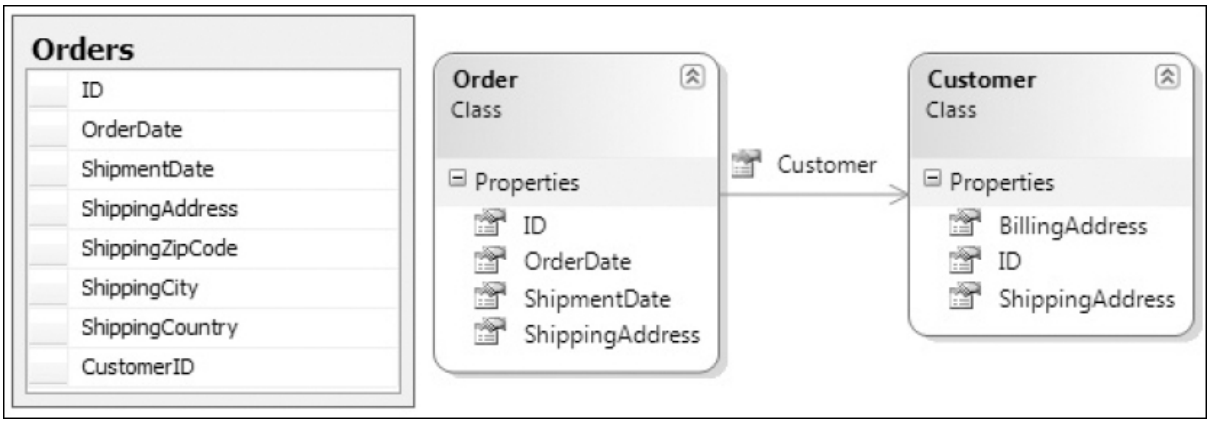


Figura 10.2 – La relazione tra ordini e clienti è mantenuta con una foreign key sul database e con una proprietà sul modello.

Le differenze si amplificano ulteriormente quando usiamo l'ereditarietà. Utilizzare questa tecnica nel mondo a oggetti è una cosa normalissima.

Tuttavia, nel mondo relazionale non esiste il concetto di ereditarietà. Supponendo di avere un modello con le classi `Customer` e `Supplier` che ereditano dalla classe `Company`, come possiamo avere una simile rappresentazione nel database? Possiamo sicuramente creare degli artifici che ci permettano poi di ricostruire le classi, ma si tratta comunque di accorgimenti volti a coprire una diversità di fondo tra il mondo relazionale e quello a oggetti.

Risolvere manualmente tutte queste complessità (e anche altre) non è affatto banale ed è per questo motivo che, da tempo, esistono dei framework che coprono queste e altre necessità. Questi framework prendono il nome di O/RM, in quanto lavorano come (M)apper tra (O)ggetti e dati (R)elazionali. In questo modo le diversità tra i due mondi sono gestite dall'O/RM, lasciandoci liberi di preoccuparci del solo codice di business.

Gli O/RM agiscono come mapper, cioè mappano le classi e le relative proprietà con le tabelle e le colonne nel database. Il vantaggio che ne deriva è nel fatto che possiamo evitare di scrivere query verso il database, ma possiamo scriverle verso l'Object Model in un linguaggio specifico dell'O/RM, che poi si preoccuperà di creare il codice SQL necessario. In termini di logica di business questo rappresenta un enorme vantaggio, in quanto gli oggetti rappresentano la logica in maniera molto più semplice delle tabelle. Lo stesso ragionamento vale per gli aggiornamenti sul database. Noi ci preoccupiamo solo di modificare gli oggetti e poi demandiamo all'O/RM la persistenza di questi sul database. Ora dovrebbe essere più chiaro cosa significhi avere un'applicazione disaccoppiata dal database.

In conclusione, un O/RM è una parte di software molto potente ma, allo stesso tempo, molto complessa e pericolosa, poiché il livello di astrazione che introduce rischia di farci dimenticare che c'è un database, e questo è negativo. Dobbiamo sempre controllare le query generate dall'O/RM e verificare le istruzioni di manipolazione dati, per essere sicuri che le performance corrispondano ai requisiti.

Ora che abbiamo capito quali compiti svolge un O/RM possiamo vedere come questi compiti siano svolti da Entity Framework Core e come possiamo usare questo strumento per semplificare lo sviluppo del codice di accesso ai dati. Il primo passo consiste nel creare le classi del modello a oggetti per poi mapparle sul database.

Mappare il modello a oggetti sul database

Visto che la M di O/RM sta per Mapper, è facile immaginare che la fase di mapping tra il modello a oggetti e il database sia molto importante. Per mappare le classi verso il database, dobbiamo svolgere tre attività: prima scriviamo le classi, poi creiamo la classe di contesto (semplicemente contesto d'ora in poi) e quindi, tramite quest'ultima, mappiamo le classi.

L'operazione di mapping viene effettuata tramite codice, utilizzando o specifiche API o decorando con attributi le classi e le proprietà dell'object model. Inoltre, se scriviamo i nomi delle classi, delle loro proprietà e delle proprietà del contesto sfruttando determinate convenzioni, non abbiamo nemmeno la necessità di scrivere il codice di mapping, in quanto Entity Framework Core è già in grado di dedurre automaticamente dai nomi come il modello debba essere mappato. Le API permettono di effettuare qualunque tipologia di mapping che Entity Framework Core metta a disposizione, mentre gli attributi e le convenzioni permettono di mappare solo un sottoinsieme delle possibilità di Entity Framework Core. Per questo motivo parleremo in maniera approfondita solo del mapping tramite API.

Vediamo ora come creare e mappare le classi introdotte nelle precedenti sezioni.

Disegnare le classi

Scrivere una classe dell'object model è estremamente semplice in quanto consiste nel creare una classe con delle proprietà, esattamente come faremmo per qualunque altra classe. Non è richiesta alcuna integrazione con Entity Framework Core, come possiamo notare nel codice dell'Esempio 10.1.

Esempio 10.1

```
public class AddressInfo
{
  public string Address { get; set; }
  public string City { get; set; }
  public string Region { get; set; }
  public string PostalCode { get; set; }
  public string Country { get; set; }
}

public partial class Customer
{
```

```
    public Customer()
    {
      Orders = new HashSet<Order>();
    }
    public string CustomerId { get; set; }
    public string CompanyName { get; set; }
    public string ContactName { get; set; }
    public string ContactTitle { get; set; }
    public AddressInfo Address { get; set; }
    public string Phone { get; set; }
    public string Fax { get; set; }
    public ICollection<Order> Orders { get; set; }
  }
  public class Order
  {
    public Order()
    {
      OrderDetails = new HashSet<OrderDetail>();
    }
    public int OrderId { get; set; }
    public string CustomerId { get; set; }
    public int? EmployeeId { get; set; }
    public DateTime? OrderDate { get; set; }
    public DateTime? RequiredDate { get; set; }
    public DateTime? ShippedDate { get; set; }
    public int? ShipVia { get; set; }
    public decimal? Freight { get; set; }
    public string ShipName { get; set; }
    public AddressInfo ShipAddress { get; set; }
    public Customer Customer { get; set; }
    public ICollection<OrderDetail> OrderDetails { get; set; }
  }
  public partial class OrderDetail
  {
    public int OrderId { get; set; }
    public int ProductId { get; set; }
    public decimal UnitPrice { get; set; }
    public short Quantity { get; set; }
    public float Discount { get; set; }
    public Order Order { get; set; }
  }
```

Il codice mostra chiaramente alcune caratteristiche del nostro modello:

- le classi sono semplici classi POCO (Plain Old CLR Object), con proprietà che rappresentano i dati e nessuna relazione con l'O/RM. Questo modello potrebbe essere usato da Entity Framework Core così come da altri O/RM senza bisogno di alcuna modifica;

- le relazioni sono espresse tramite proprietà (dette Navigation Property) che si riferiscono direttamente a un oggetto in caso di

relazioni uno a uno (come da dettaglio a ordine), o una lista di oggetti in casi di relazioni uno a molti (come da ordine a dettagli);

- il tipo `AddressInfo` è un tipo senza una chiave, referenziato da altri oggetti. Questo tipo di classe è detta owned type e agisce come semplice contenitore di proprietà e non come tipo da mappare verso una tabella;

- anche se non presenti nell'esempio, gli `enum` sono supportati come qualunque altro tipo nativo.

Ora che abbiamo visto il codice dell'object model andiamo a vedere il codice del contesto.

Creare il contesto

Il contesto è la classe che agisce da ponte tra il mondo a oggetti dell'object model e il mondo relazionale del database. Infatti, è attraverso questa classe che possiamo mappare l'object model verso il database ed effettuare tutte le operazioni, siano esse query o modifiche di dati negli oggetti.

Il contesto è una classe che eredita da **`DbContext`** e che definisce una proprietà di tipo `DbSet<T>`, chiamata entityset, per ogni classe dell'object model mappata verso una tabella del database (il tipo `T` corrisponde al tipo della classe). Nel nostro caso, il contesto conterrà tre proprietà: una per la classe `Customer`, una per la classe `Order` e una per la classe `Order_Detail`. Queste proprietà rappresentano il punto di entrata per recuperare e modificare oggetti nel database.

> *Nel caso in cui usiamo l'ereditarietà nel nostro object model (per esempio una classe Company da cui derivano Customer e Supplier), abbiamo un solo entityset per tutta la gerarchia. Il tipo dell'entityset è quello della classe base.*

Oltre a queste proprietà, il contesto definisce anche un costruttore che accetta in input un oggetto `DbContextOptions<T>`, dove `T` è il tipo del contesto, che viene passato in input al costruttore base. Questo costruttore è fondamentale per l'integrazione con ASP.NET come vedremo in seguito. L'Esempio 10.2 mostra il codice del contesto.

Esempio 10.2

```
public class NorthwindContext : DbContext
{
  public NorthwindContext(DbContextOptions<NorthwindContext> o) : base(o) { }

  public DbSet<Customer> Customers { get; set; }
  public DbSet<OrderDetail> OrderDetails { get; set; }
  public DbSet<Order> Orders { get; set; }

  protected override void OnConfiguring(DbContextOptionsBuilder o)
  {
  }
}
```

Possiamo configurare la classe di contesto eseguendo l'override del metodo `OnConfiguring`, che accetta un oggetto di tipo **`DbContextOptionsBuilder`**, che espone i parametri di configurazione. Ora che abbiamo visto come creare la classe di contesto, vediamo come eseguire il mapping delle classi verso il database attraverso questa classe.

Mapping tramite convenzioni

Quando un contesto viene istanziato la prima volta e viene eseguita la prima operazione, Entity Framework Core esegue il codice di mapping. Per prima cosa vengono analizzate le proprietà di tipo `DbSet<T>` del contesto per recuperarne le relative classi e verificare se queste rispettino determinate convenzioni che ne permettono il mapping senza necessità di scrivere codice. Per chiarire meglio questo concetto, analizziamo le convenzioni applicate alla classe `Order`:

- La classe viene mappata verso una tabella che ha il nome dell'entityset. Nel nostro caso, la classe viene mappata verso la tabella `Orders`;

- le colonne della tabella hanno lo stesso nome delle proprietà della classe. Nel caso di proprietà all'interno di owned type, il nome del campo sulla tabella viene calcolato unendo il nome della proprietà di tipo owned type con quello della proprietà al suo interno e separandoli con il carattere "_". Nel nostro caso, la proprietà `City` all'interno della proprietà `ShipAddress` viene mappata sulla colonna

ShipAddress_City. Se un owned type contiene un altro owned type, la tecnica di calcolo non cambia e i nomi vengono concatenati;

- se una proprietà si chiama, indipendentemente dal case, ID o {NomeClasse}ID (dove il segnaposto NomeClasse viene rimpiazzato dal nome della classe che contiene la proprietà) questa viene automaticamente eletta a chiave primaria della classe. Se la proprietà è di tipo intero, questa è trattata come Identity. Nel nostro caso, la proprietà OrderId è automaticamente identificata come chiave primaria;

- Il tipo del campo su cui la proprietà è mappata è analogo al tipo della proprietà (int per i tipi Int32, bit per i tipi Boolean e così via). Le proprietà di tipo Nullable<T> e le proprietà di tipo String sono considerate null sul database, le altre proprietà sono considerate not null. Infine, le proprietà di tipo String sono considerate unicode a lunghezza massima. Nel nostro caso, la proprietà CustomerId è mappata su una colonna di tipo int, mentre la proprietà ShipName è mappata su una colonna di tipo nvarchar;

- Le navigation property con una reference uno a uno vengono automaticamente mappate utilizzando come nome della colonna il nome della proprietà chiave della classe a cui la proprietà si riferisce. Nel nostro caso, la navigation property Customer punta a un oggetto di tipo Customer la cui proprietà chiave è CustomerId. Questo significa che Entity Framework Core sfrutta la colonna CustomerId nella tabella Orders per mappare la relazione tra le due entità.

Alla luce di queste convenzioni appare chiaro che buona parte del codice di mapping può essere gestito automaticamente dalle convenzioni. Tuttavia c'è una parte di mapping che va gestita a mano come la lunghezza massima delle stringhe, la configurazione dei nomi dei campi quando sono diversi dalle proprietà, la configurazione di chiavi primarie composte da più proprietà, gli indici, le foreign key e altro ancora.

Mapping tramite API

Per mappare una classe verso il database usando le API di Entity Framework Core, dobbiamo eseguire l'override del metodo `OnModelCreating` nella classe di contesto. Questo metodo accetta in input un oggetto di tipo **`ModelBuilder`**. Il metodo principale di questa classe è `Entity`, che accetta il tipo della classe come parametro generico e un metodo che specifica il mapping della classe. Questo metodo prende in input un oggetto di tipo `EntityTypeBuilder<T>`, dove `T` rappresenta la classe, tramite cui possiamo eseguire tutte le operazioni di mapping della classe. I metodi principali della classe `EntityTypeBuilder<T>` sono esposti nella seguente tabella.

Tabella 10.1 – Metodi di mapping di EntityTypeBuilder<T>.

Metodo	Scopo
`Property`	Accetta una lambda che rappresenta una proprietà della classe e restituisce un oggetto che la rappresenta e sul quale possiamo eseguire metodi per mapparne le caratteristiche (nome della colonna sulla tabella, dimensioni, precisione, nullabilità e così via)
`HasIndex`	Accetta una lambda che rappresenta una o più proprietà che formano un indice sulla tabella e restituisce un oggetto su cui applicare metodi di mapping per specificare alcune caratteristiche dell'indice (nome, univocità)
`ToTable`	Specifica il nome della tabella su cui la classe viene mappata nel database
`OwnsOne`	Accetta una lambda che rappresenta una proprietà della classe che si riferisce a un owned type e restituisce un oggetto che la rappresenta. Su questo oggetto possiamo usare metodi per mappare le proprietà dell'owned type. Questi metodi sono quasi gli stessi visti in questa tabella.
`HasOne/HasMany`	Accettano una lambda che rappresenta una navigation property della classe e restituisce un oggetto che permette di specificarne le caratteristiche di mapping (nome della colonna sul db, relazione con la classe corrispondente, nome della foreign key).
`HasFilter`	Accetta una lambda che specifica un filtro che verrà applicato a ogni query che riguarda la tabella, senza bisogno di specificarlo in ogni query
`HasKey`	Accetta una lambda che rappresenta una o più proprietà che formano la chiave primaria della tabella e restituisce un oggetto su cui applicare metodi di mapping per specificare alcune caratteristiche della chiave (nome, clustered se il database è SqlServer)

I metodi di `EntityTypeBuilder<T>` si dividono in due categorie: quelli che mappano informazioni tra la classe e la tabella (`HasKey`, `ToTable`, `HasFilter`) e quelle che tornano le proprietà per poi specificarne il mapping (`Property`, `OwnsOne`, `HasOne`, `HasMany`). Vediamo nella prossima tabella quali sono i metodi di mapping relativi alle proprietà.

Tabella 10.2 – Metodi di mapping.

Metodo	Applicabile su	Scopo
`HasMaxLength`	`String`	Specifica la lunghezza massima del campo
`IsUnicode`	`String`	Specifica se il campo supporta caratteri unicode
`HasColumnName`	Tutti i tipi	Specifica il nome del campo mappato
`HasColumnType`	Tutti i tipi	Specifica il tipo del campo mappato
`IsRequired`	Tutti i tipi	Specifica se il campo è null o no (not null per default)
`IsConcurrencyToken`	Tutti i tipi	Specifica che la proprietà fa parte del token per gestire la concorrenza ottimistica negli aggiornamenti
`IsRowVersion`	Tutti i tipi	Specifica che la proprietà contiene la versione della riga (ogni database può interpretare il campo mappato in modo diverso)
`ValueGeneratedNever`	Tutti i tipi	Specifica che il valore della proprietà viene generato dal client (la nostra applicazione)
`ValueGeneratedOnAdd`	Tutti i tipi	Specifica che il valore della proprietà può essere generato dal client in fase di inserimento, ma spetta al database decidere se usare quello del client o generarne un altro. Per fare un esempio, se usiamo un'identity con SqlServer, un eventuale valore fornito dal client viene scartato e sostituito con quello generato dal server.
`ValueGeneratedOnAddOrUpdate`	Tutti i tipi	Specifica che il valore della proprietà può essere generato dal client sia in fase di inserimento sia in fase di aggiornamento. Così come per `ValueGeneratedOnAdd`, spetta al database decidere se usare il valore inviato dal del client o generarne un altro.
`HasDefaultValueSql`	Tutti i tipi	Specifica il valore di default della proprietà sul database.

I metodi di mapping appena elencati sono agnostici rispetto al tipo di database. Il provider di Entity Framework Core per SqlServer aggiunge ulteriori metodi specifici per il database (identity, sequence e altro). Altri provider per altri database possono aggiungere altri metodi (tramite extension method) per consentire altre modalità di mapping specifiche per le loro esigenze.

Ora che abbiamo illustrato i metodi di mapping, vediamo come applicarli per mappare la classe `AddressInfo` all'interno del metodo `OnModelCreating`.

Esempio 10.3

```
modelBuilder.Entity<AddressInfo>(entity =>
{
  entity.Property(e => e.City).HasMaxLength(15);
  entity.Property(e => e.Country).HasMaxLength(15);
  entity.Property(e => e.PostalCode).HasMaxLength(10);
  entity.Property(e => e.Region).HasMaxLength(15);
  entity.Property(e => e.Address).HasMaxLength(60);
});
```

In questo esempio sfruttiamo il metodo `Property` per recuperare il riferimento a una proprietà dell'owned type e specifichiamo la massima lunghezza sfruttando il metodo `HasMaxLength`. Il tipo `AddressInfo` è piuttosto semplice da mappare, quindi possiamo passare al prossimo: `Customer`.

Esempio 10.4

```
modelBuilder.Entity<Customer>(entity =>
{
  entity.HasKey(e => e.CustomerId);

  entity.Property(e => e.CustomerId)
    .HasColumnType("nchar(5)")
    .ValueGeneratedNever();

  entity.HasIndex(e => e.CompanyName)
    .HasName("CompanyName");

  entity.OwnsOne(e => e.Address)
    .Property(e => e.Address)
    .HasColumnName("Address");
```

```
    entity.OwnsOne(e => e.Address)
      .Property(e => e.City)
      .HasColumnName("City");

    entity.OwnsOne(e => e.Address)
      .Property(e => e.Country)
      .HasColumnName("Country");

    entity.OwnsOne(e => e.Address)
      .Property(e => e.PostalCode)
      .HasColumnName("PostalCode");

    entity.OwnsOne(e => e.Address)
      .Property(e => e.Region)
      .HasColumnName("Region");

    entity.OwnsOne(e => e.Address)
      .HasIndex(e => e.City)
      .HasName("City");

    entity.OwnsOne(e => e.Address)
      .HasIndex(e => e.PostalCode)
      .HasName("PostalCode");

    entity.Property(e => e.CompanyName)
      .IsRequired()
      .HasMaxLength(40);

    entity.Property(e => e.ContactName).HasMaxLength(30);
    entity.Property(e => e.ContactTitle).HasMaxLength(30);
    entity.Property(e => e.Fax).HasMaxLength(24);
    entity.Property(e => e.Phone).HasMaxLength(24);
  });
```

La classe `Customer` è decisamente più complessa rispetto a `AddressInfo`. Innanzitutto specifichiamo che la proprietà che mappa sulla chiave primaria è `CustomerId`. Questo mapping non sarebbe necessario in quanto Entity Framework Core è in grado di capirlo dalle convenzioni, ma è comunque riportato per dare un esempio del suo utilizzo.

Successivamente alla specifica della chiave primaria, viene specificato il mapping della proprietà che agisce come chiave primaria sfruttando il metodo `HasColumnType`, per indicare il tipo del campo sulla tabella, e `ValueGeneratedNever` per indicare che il valore viene sempre generato dal client. Nell'istruzione che segue, tramite il metodo `HasIndex`, specifichiamo che la tabella ha un indice, che la colonna `CompanyName` fa parte dell'indice e che l'indice si chiama come la colonna (metodo `HasName`). Possiamo specificare che un indice è univoco aggiungendo la chiamata al metodo `IsUnique`.

Le istruzioni successive mappano le proprietà dell'owned type `AddressInfo` verso la tabella `Customers`. Qui è interessante il modo in cui recuperiamo le proprietà. Questo recupero potrebbe avvenire con una unica lambda, ma, poiché Entity Framework Core non la supporta,

dobbiamo prima recuperare la proprietà `Address` all'interno di `Customer` e poi recuperare le sue proprietà interne, per mapparle verso la relativa colonna con il metodo `HasColumnName`. Quest'operazione è necessaria in quanto in sua assenza Entity Framework Core utilizzerebbe le convenzioni e userebbe i nomi `Address_City`, `Address_Address` e così via, come nomi delle colonne sul database, generando quindi un'eccezione a run time, in quanto queste colonne non esistono.

Oltre a specificare i nomi delle colonne che mappano sulle proprietà dell'owned type, specifichiamo anche gli indici presenti su tali colonne. Infine, usiamo i metodi `IsRequired` e di nuovo `HasMaxLength` per specificare le caratteristiche delle proprietà rimaste.

Ora cambiamo classe e analizziamo il mapping di `Order` che viene mostrato nell'Esempio 10.5.

Esempio 10.5

```
modelBuilder.Entity<Order>(entity =>
{
  entity.HasKey(e => e.OrderId);
  entity.HasIndex(e => e.CustomerId).HasName("CustomersOrders");
  entity.HasIndex(e => e.EmployeeId).HasName("EmployeesOrders");
  entity.HasIndex(e => e.OrderDate).HasName("OrderDate");

  entity.OwnsOne(e => e.ShipAddress)
    .HasIndex(e => e.PostalCode)
    .HasName("ShipPostalCode");

  entity.OwnsOne(e => e.ShipAddress)
    .Property(e => e.Address)
    .HasColumnName("ShipAddress");

  entity.OwnsOne(e => e.ShipAddress)
    .Property(e => e.City)
    .HasColumnName("ShipCity");

  entity.OwnsOne(e => e.ShipAddress)
    .Property(e => e.Country)
    .HasColumnName("ShipCountry");

  entity.OwnsOne(e => e.ShipAddress)
    .Property(e => e.PostalCode)
    .HasColumnName("ShipPostalCode");

  entity.OwnsOne(e => e.ShipAddress)
    .Property(e => e.Region)
    .HasColumnName("ShipRegion");

  entity.HasIndex(e => e.ShipVia)
    .HasName("ShippersOrders");

  entity.HasIndex(e => e.ShippedDate)
    .HasName("ShippedDate");

  entity.Property(e => e.OrderId).HasColumnName("OrderID");
  entity.Property(e => e.EmployeeId).HasColumnName("EmployeeID");
  entity.Property(e => e.Freight)
```

```
      .HasColumnType("money")
      .HasDefaultValueSql("((0))");

    entity.Property(e => e.OrderDate).HasColumnType("datetime");
    entity.Property(e => e.RequiredDate).HasColumnType("datetime");
    entity.Property(e => e.ShipName).HasMaxLength(40);
    entity.Property(e => e.ShippedDate).HasColumnType("datetime");
    entity.HasOne(d => d.Customer).WithMany(p => p.Orders);
  });
```

Il mapping della classe `Order` utilizza molti dei metodi già visti nei precedenti esempi. Infatti, vengono prima specificati gli indici, quindi mappate le proprietà dell'indirizzo e poi mappate le proprietà rimanenti. In particolare, per la proprietà `Freight` viene usato il metodo `HasDefaultValueSql`, il quale specifica il valore di default del campo sulla tabella del database.

Un altro metodo interessante è `HasOne`. Questo viene utilizzato per mappare la navigation property `Customer` verso l'omonima classe. Per convenzione, Entity Framework Core usa `CustomerId` come nome della colonna che agisce da foreign key verso la tabella `Customers`. Questo perché la colonna che rappresenta la chiave primaria sulla tabella `Customers` si chiama `CustomerId`. Possiamo modificare questo comportamento usando il metodo `HasForeignKey` e passando in input il nome della colonna sulla tabella `Orders`. Il successivo metodo `WithMany` specifica che la navigation property `Orders` di `Customer` è mappata verso `Order` sfruttando la stessa foreign key.

Ora non rimane che mappare la classe `OrderDetail`.

Esempio 10.6

```
  modelBuilder.Entity<OrderDetail>(entity =>
  {
    entity.ToTable("Order Details");
    entity.HasKey(e => new { e.OrderId, e.ProductId });
    entity.HasIndex(e => e.OrderId).HasName("OrdersOrder_Details");
    entity.HasIndex(e => e.ProductId).HasName("ProductsOrder_Details");
    entity.Property(e => e.OrderId).HasColumnName("OrderID");
    entity.Property(e => e.ProductId).HasColumnName("ProductID");
    entity.Property(e => e.Discount).HasDefaultValueSql("((0))");
    entity.Property(e => e.Quantity).HasDefaultValueSql("((1))");
    entity.Property(e => e.UnitPrice)
      .HasColumnType("money")
      .HasDefaultValueSql("((0))");

    entity.HasOne(d => d.Order)
      .WithMany(p => p.OrderDetails)
      .OnDelete(DeleteBehavior.ClientSetNull)
  });
```

Il mapping di `OrderDetail` utilizza altri metodi molto importanti.

Il primo è `ToTable` che specifica il nome della tabella verso cui la classe mappa. In questo caso, siamo obbligati a usare questo metodo in quanto il nome della tabella dedotto dalle convenzioni, `OrderDetails`, è errato.

Il secondo è `HasKey`. Sebbene abbiamo già visto questo metodo prima, in questo caso il suo utilizzo mostra come specificare una chiave composta da più proprietà. Infatti, il metodo non ritorna una sola proprietà, bensì un anonymous type con le proprietà che fanno parte della chiave primaria. Questa stessa tecnica è utilizzabile anche nel metodo `HasIndex` per specificare indici composti.

Come abbiamo detto in precedenza, oltre che tramite convenzioni e API, esiste un terzo metodo di mapping che utilizza le data annotation e di cui ci occuperemo nella prossima sezione.

Mapping tramite data annotation

Il mapping tramite le data annotation prevede che in testa alle classi dell'object model e alle proprietà siano presenti alcuni attributi che vengono interpretati dal motore di Entity Framework Core. Questi attributi permettono di specificare il nome della tabella su cui una classe mappa, il nome della colonna su cui una proprietà mappa, il tipo specifico della colonna, se è una primary key, se è obbligatoria, la sua lunghezza massima e altro ancora. Sebbene sia comodo, il mapping tramite data annotation non copre tutte le esigenze, come invece fa il mapping tramite API. La Tabella 10.3 mostra le principali data annotation.

Tabella 10.3 – Data annotation per il mapping.

Attributo	Applicabile su	Scopo
`Table`	Classe	Specifica la tabella su cui la classe mappa
`Key`	Proprietà	Specifica che la proprietà fa parte della chiave primaria
`Column`	Proprietà	Specifica il nome e il tipo della colonna su cui la proprietà mappa
`DatabaseGenerate`	Proprietà	Specifica se la colonna è su cui la proprietà mappa

d		è un'identity, calcolata o normale
Required	Proprietà	Specifica che la proprietà è obbligatoria
StringLength	Proprietà	Specifica la lunghezza massima della proprietà

A prescindere dalla tecnica di mapping utilizzata, abbiamo scritto il codice necessario per cominciare a utilizzare Entity Framework Core. Tuttavia, finora siamo ricorsi alla scrittura manuale del codice, ma Entity Framework Core permette di generare il codice partendo dalle tabelle del database.

Generare automaticamente le classi dal database

Quando abbiamo già il database a disposizione e dobbiamo generare l'object model, possiamo utilizzare il comando `scaffold-dbcontext`. Questo comando prende in input la stringa di connessione al database, il provider di Entity Framework da usare e genera una classe per ogni tabella, mappando automaticamente le proprietà e le navigation property, e generando anche la classe di contesto con tutti gli entityset.

> *Oltre ai parametri base, possiamo specificare anche di quali tabelle vogliamo eseguire il mapping, in quale cartella mettere i file generati, il nome del contesto, se generare un log di generazione e altro ancora. Per conoscere tutti i parametri, rimandiamo alla pagina della documentazione, disponibile all'url:* http://aspit.co/bk0.

Nell'Esempio 10.7 possiamo vedere il comando da utilizzare per generare le classi, il contesto e il mapping dell'object model che abbiamo analizzato.

Esempio 10.7

```
scaffold-dbcontext
  -connection "connection string"
  -provider "Microsoft.EntityFrameworkCore.SqlServer"
  -tables "Customers", "Order Details", "Orders"
  -output "data"
  -context "NorthwindContext"
  -verbose
```

Questo comando va lanciato dalla finestra Package Manager Console di Visual Studio, all'interno della quale va prima selezionato su quale progetto creare i file e poi lanciato il comando.

Se non usiamo Visual Studio, possiamo impiegare le estensioni di Entity Framework Core per la CLI di .NET: `dotnet ef`. Queste estensioni sono installate globalmente, quindi non necessitano di pacchetti NuGet da installare.

Esempio 10.8

```
dotnet ef dbcontext scaffold
  ″connection string″
  Microsoft.EntityFramework.SqlServer
  -t ″Customers″, ″Order Details″, ″Orders″
  -o ″data″
  -c ″NorthwindContext″
  -v
```

Il generatore crea classi e proprietà mappandole uno a uno con le tabelle e le colonne del database. Questo significa che non prende in considerazione la creazione di owned type come `AddressInfo`.

> *Oltre al comando per generare le classi e il contesto partendo dal database, ne esistono molti altri. Esistono comandi per manipolare e applicare le migrazioni, per eliminare il database e per ottenere informazioni sui contesti presenti nel progetto.*

Avere a disposizione il codice di mapping (sia esso scritto a mano o generato da Entity Framework Core) è la prima parte di ciò che serve per far funzionare Entity Framework Core; la seconda è la configurazione del contesto in ASP.NET.

Configurare Entity Framework Core con ASP.NET

Come abbiamo visto nei precedenti capitoli, ASP.NET è fortemente basato sulla dependency injection. Poiché il contesto è una dipendenza dei controller, possiamo aggiungerlo come parametro del costruttore del controller e lasciare che sia il motore di dependency injection di ASP.NET a gestirne il ciclo di vita. La configurazione del contesto avviene nel metodo

ConfigureServices della classe Startup, all'interno del quale usiamo il metodo AddDbContext. Questo metodo accetta in input una funzione, la quale accetta un parametro che rappresenta le opzioni del contesto e configura tali opzioni. Quelle principali sono il tipo di database e la stringa di connessione, senza i quali non sarebbe possibile utilizzare il contesto. Queste opzioni vengono poi passate al costruttore del contesto. L'Esempio 10.9 mostra il codice necessario alla configurazione.

Esempio 10.9

```
services.AddDbContext<NorthwindContext>(o =>
{
  var connectionString = ″stringa di connessione da configurazione″;
  o.UseSqlServer(connectionString);
});
```

Il metodo UseSqlServer specifica che intendiamo usare il provider di Entity Framework Core per SqlServer e la stringa in input rappresenta la stringa di connessione da usare. Per default, il metodo registra il contesto come Scoped così che esista una sola istanza per richiesta web. Volendo possiamo modificare questo comportamento passando un secondo parametro al metodo AddDbContext di tipo ServiceLifetime e specificando un lifetime diverso.

> *Avere un'istanza del contesto per richiesta è il pattern più comune. A seconda delle esigenze, si può preferire gestire manualmente il ciclo di vita del contesto per avere più istanze in una richiesta, ma non si deve mai avere un contesto singletone perchè questo verrebbe condiviso da tutti gli utenti dell'applicazione, generando comportamenti imprevedibili.*

Al posto del metodo AddDbContext possiamo usare anche AddDbContextPool. Questo metodo permette una piccola ottimizzazione di performance. Con AddDbContext, ASP.NET crea ogni volta un'istanza nuova. Sebbene l'istanziazione di un contesto sia abbastanza leggera, ha comunque un costo. Il metodo AddDbContextPool genera un pool di contesti e ogni volta che ASP.NET deve generarne uno, invece che crearlo da zero lo va a prendere dal pool. In questo modo abbiamo una maggior

lentezza in fase di startup ma una maggiore velocità durante l'esecuzione delle richieste. L'unica accortezza da mantenere è quella di non salvare alcuno stato nel contesto, altrimenti quello stato verrà usato anche dalle richieste che successivamente sfrutteranno quella specifica istanza del contesto. La firma di `AddDbContextPool` è la stessa di `AddDbContext`, quindi il suo utilizzo è estremamente semplice.

A questo punto siamo pronti per utilizzare Entity Framework Core.

Recuperare i dati dal database

La ricerca di dati nel database avviene tramite gli entityset del contesto, quindi la prima cosa da fare è recuperare l'istanza del contesto. Grazie alla sua configurazione nel sistema di dependency injection di ASP.NET, possiamo ottenerlo in input nel costruttore oppure come parametro della action, aggiungendo al parametro l'attributo `FromServices`.

Esempio 10.10

```
public class HomeController
{
  NorthwindContext _ctx;
  public HomeController(NorthwindContext ctx) {
    _ctx = ctx;
  }
}
public class HomeController
{
  public IActionResult Index([FromServices] NorthwindContext ctx)
  {
  }
}
```

Ora che abbiamo il contesto, dobbiamo sfruttare i suoi entityset. Poiché il tipo `DbSet<T>` implementa indirettamente l'interfaccia `IQueryable<T>`, questo espone tutti gli extension method di LINQ. Questo non significa assolutamente che i dati siano in memoria. Il provider LINQ di Entity Framework Core intercetta l'esecuzione della query verso l'entityset e la trasforma in un *expression tree,* che viene poi convertito in SQL grazie alle informazioni di mapping. Il risultato è che noi scriviamo query LINQ e tutto il lavoro di conversione è affidato a Entity Framework Core.

L'Esempio 10.11 mostra come sia semplice recuperare la lista dei clienti italiani.

Esempio 10.11

```
var customers1 = from c in ctx.Customers
                 where c.Address.Country == "Italy"
                 select c;

var customers2 = ctx.Customers.Where(c => c.Address.Country == "Italy");
```

L'Esempio 10.11 porta ad alcune importanti considerazioni. La prima è che non dobbiamo gestire né connessioni né transazioni né altri oggetti legati all'interazione col database. Entity Framework Core astrae tutto per noi, restituendoci direttamente oggetti.

La seconda è che il codice che abbiamo appena visto non esegue alcuna query. La deferred execution è perfettamente supportata dal provider LINQ di Entity Framework Core, quindi, finché non enumeriamo fisicamente i risultati, la query non viene effettuata sul database. È importante sottolineare che a causa di questo comportamento, se eseguiamo la query quando il contesto è stato eliminato, otteniamo un'eccezione a run time. Il caso più comune in cui ci troviamo in questa situazione è quando assegniamo la variabile `customer1` a una proprietà del modello e poi nella view enumeriamo la proprietà. La query viene scatenata quando enumeriamo la proprietà ma, quando siamo nel contesto di esecuzione della view, il nostro contesto è già stato eliminato, in quanto siamo usciti dal suo scope.

La terza cosa da notare è che nelle query possiamo utilizzare sia la query syntax sia la normale sintassi basata sugli extension method, ma nel secondo caso dobbiamo prestare attenzione a quali extension method utilizzare. Il provider di Entity Framework Core non supporta tutti i metodi LINQ e i relativi overload poiché alcuni non trovano una controparte sui database, mentre altri non hanno la possibilità di essere tradotti in SQL. Gli extension method disponibili sono quelli di aggregazione, di intersezione, di ordinamento, di partizionamento, di proiezione, di raggruppamento oltre a quelli di insieme.

Per esempio, quando vogliamo recuperare un solo oggetto, i metodi `First` e `Single` sono validi. La differenza risiede nel fatto che mentre `First`

viene tradotto in una `TOP 1`, `Single` viene tradotto in una `TOP 2`, per verificare che non sia presente più di un elemento. Inoltre se la ricerca dell'oggetto avviene per chiave primaria, possiamo anche utilizzare il metodo `Find` (o la sua controparte asincrona `FindAsync`) della classe `DbSet<T>` come mostrato nel prossimo esempio.

Esempio 10.12

```
var c1 = ctx.Customers.First(c => c.CustomerID == "ALFKI");
var c2 = ctx.Customers.Find("ALFKI");
var c3 = await ctx.Customers.FindAsync("ALFKI");
```

Una cosa che torna utilissima in LINQ sono le proiezioni. Molto spesso non abbiamo bisogno di tutta la classe, ma solo di alcuni suoi dati. Specificando in una proiezione quali proprietà dobbiamo estrarre (come nell'Esempio 10.13), faremo in modo che l'SQL generato estragga solo quelle, ottenendo così un'ottimizzazione delle performance.

Esempio 10.13

```
ctx.Customers.Select(c => new { c.CustomerID, c.CompanyName});
```

Esempio 10.13 – SQL

```
SELECT
  c.CustomerID,
  c.CompanyName
FROM [dbo].[Customers] AS c
```

Il provider LINQ di Entity Framework Core è un argomento controverso. Da un lato, le potenzialità del provider sono enormi, in quanto traduce molti metodi LINQ in SQL, supporta l'esecuzione mista di codice LINQ che genera SQL e codice LINQ eseguito in locale, permette di mischiare parzialmente codice SQL e codice LINQ per generare una unica query SQL sul server, e altro ancora. Dall'altro lato, alcuni metodi di LINQ non sono supportati, o sono solo parzialmente supportati, mentre altri si comportano in modo inaspettato, con conseguenze non facili da identificare specie se si è alle prime armi. Fortunatamente, Entity Framework Core permette di lanciare

un'eccezione ogni volta che esegue una query che scarica i dati in locale e li processa invece di generare codice SQL. L'Esempio 10.14 mostra come fare.

Esempio 10.14

```
protected override void OnConfiguring(DbContextOptionsBuilder ob)
{
  ob.ConfigureWarnings(warnings =>
    warnings.Throw(RelationalEventId.QueryClientEvaluationWarning));
}
```

Il metodo `ConfigureWarnings` accetta un delegato che viene invocato ogni volta che il motore solleva un warning. Nel delegato specifichiamo che quando il tipo di warning è relativo a una valutazione della query sul client (ovvero i dati vengono scaricati sul client e poi processati) deve essere sollevata un'eccezione. In questo modo possiamo evitare possibili problemi di performance.

In altri casi, le query possono soffrire del problema 1+n, rallentando vistosamente le performance dell'applicazione. Un caso in cui questo accade è quando usiamo una projection che include campi di una navigation property che punta a una lista di oggetti.

Prendiamo in esame il seguente codice.

Esempio 10.15

```
var orders = ctx.Orders
  .Where(o => o.Customer.CustomerId == "ALFKI")
  .Select(c => new {
    CustomerId = c.Customer.CustomerId,
    Products = c.OrderDetails.Select(d => d.ProductId)
  })
  .ToList (); // query che estrae dati ordine
foreach (var order in orders)
{
  foreach (var p in order.Products) // query che estrae prodotti per ordine
  {
  }
}
```

La query filtra gli ordini e per ogni ordine estrae l'id del cliente e gli id dei prodotti coinvolti nell'ordine. In questi casi ci si aspetta che venga eseguita una sola query che estrae i dati utilizzando una `join` SQL. In realtà, il

motore esegue prima una query che estrae solo i dati degli ordini (`CustomerId` in questo caso), e poi, quando accediamo in ciclo alla lista dei prodotti per l'ordine corrente, esegue una query per estrarre i prodotti. Questo significa che se la query torna 10 ordini, il codice esegue una query per estrarre i dati dell'ordine e poi una query per ogni ordine per estrarre i prodotti, per un totale di 11 query.

I casi appena mostrati evidenziano come sia sempre bene controllare il codice SQL generato utilizzando strumenti come SqlServer Profiler o sfruttando il logging di Entity Framework Core.

Ottimizzare il fetching

Come abbiamo visto in precedenza, quando recuperiamo gli oggetti, spesso abbiamo bisogno anche di altri oggetti collegati. Nella maggioranza dei casi, la soluzione ideale è quella di recuperare tutti gli oggetti in una singola query. Il problema risiede nel fatto che Entity Framework Core ritorna solo gli oggetti specificati dall'entityset. Questo significa che quando eseguiamo una query sugli ordini, i dettagli vengono ignorati. Fortunatamente possiamo modificare questo comportamento e dire quali dati collegati vogliamo caricare insieme all'entità principale.

Per fare questo dobbiamo utilizzare il metodo `Include` della classe `DbSet`. Questo metodo, in input, accetta una lambda che specifica la proprietà da caricare. Quando dobbiamo caricare proprietà che fanno parte dell'oggetto caricato tramite `Include`, possiamo concatenare il metodo `ThenInclude`, che accetta una lambda che specifica la proprietà da caricare. L'utilizzo di entrambi i metodi è mostrato nel seguente codice.

Esempio 10.16

```
ctx.Orders.Include(o => o.OrderDetails);
ctx.Orders.Include(o => o.OrderDetails).ThenInclude(c => c.Product);
```

Il codice SQL generato dalla prima query mette in join le tabelle `Orders` e `Order Details` per recuperare ordini e dettagli, mentre quello generato dalla seconda query mette in join `Orders` e `Order Details` e `Products` per recuperare ordini, dettagli e i prodotti legati ai dettagli.

Sebbene recuperare immediatamente i dettagli sia ottimale nella maggior parte dei casi, vi sono delle situazioni in cui è meglio recuperare solo gli ordini e accedere ai dettagli sfruttando il lazy loading.

Lazy loading

Tramite questa tecnica effettuiamo una query per recuperare una o più entity principali e ne recuperiamo le entità collegate solo se fisicamente vi accediamo. Per fare un esempio, potremmo dover recuperare tutti gli ordini di un giorno, ma recuperare i dettagli solo di quelli spediti in certi stati. In questo caso recuperiamo gli ordini in una query unica e solo quando accediamo ai dettagli viene effettuata una query da Entity Framework Core (per noi trasparente in quanto accediamo semplicemente alla proprietà `OrderDetails`). Cosa che abbiamo trattato nella precedente sezione, questo significa introdurre il problema del 1+n, poiché si esegue una query per ogni ordine di cui recuperiamo i dettagli, quindi è bene valutare l'uso del lazy loading.

Oltre a questa problematica, per supportare il lazy loading dobbiamo anche apportare modifiche al codice. Per prima cosa, dobbiamo aggiungere tramite NuGet un riferimento all'assembly `Microsoft.EntityFrameworkCore.Proxies`. Successivamente, nella configurazione del contesto dobbiamo aggiungere la chiamata al metodo `UseLazyLoadingProxies`.

Esempio 10.17

```
services.AddDbContext<NorthwindContext>(o =>
{
  o.UseLazyLoadingProxies();
  o.UseSqlServer("Data Source=(local);Initial Catalog=Northwind;
    Integrated Security=True");
});
```

Infine dobbiamo modificare le classi dell'object model, rendendole tutte non `sealed`, e impostando come `virtual` tutte le navigation property. Questa operazione è necessaria in quanto, a run time, Entity Framework Core crea un nuovo tipo (proxy) che eredita dalla nostra classe dell'object model e che sovrascrive getter e setter delle navigation property, al fine di

iniettare il codice necessario a effettuare il lazy loading, cioè andare sul database e recuperare i dati. L'Esempio 10.18 mostra come il lazy loading sia trasparente per il nostro codice.

Esempio 10.18

```
var orders = _ctx.Orders.ToList ();
foreach (var order in orders)
{
  if (order.ShipAddress.Country == "Italy")
  {
    foreach (var detail in order.OrderDetails)
    {

    }
  }
}
```

Il risultato di questo codice è che per ogni ordine spedito in Italia, noi accediamo ai dettagli ed Entity Framework Core scatena una query in maniera trasparente per noi. Se avessimo caricato tutti i dettagli per tutti gli ordini, avremmo effettuato una sola query al database (anzi due, viste le modalità di querying di Entity Framework Core), ma avremmo caricato moltissimi dati inutili. In questo modo, effettuiamo più query ma carichiamo solo i dati effettivamente utili. La scelta tra una tecnica e l'altra va valutata caso per caso.

> *In questo caso specifico, la soluzione migliore sarebbe stata quella di effettuare una query che estraesse tutti i dettagli degli ordini spediti in Italia, ma lo scopo era solo quello di mostrare il funzionamento del lazy loading.*

Lavorare con i proxy non è sempre la cosa ideale, soprattutto quando si ha a che fare con la reflection. Nei casi in cui non vogliamo usare i proxy ma vogliamo mantenere il lazy loading, possiamo iniettare in un costruttore privato della nostra classe, l'interfaccia `ILazyLoader`, oppure un delegato di tipo `Action<object, string>`, il cui nome deve essere `lazyLoader`. Grazie all'iniezione di questi oggetti, possiamo invocare il caricamento dei dati, scrivendo noi il codice senza dover ricorrere ai proxy e a tutto quello che il loro utilizzo comporta. Per maggiori informazioni su queste tecniche,

rimandiamo alla documentazione, disponibile all'indirizzo: http://aspit.co/bnu.

Le possibilità di querying non si fermano qui. Infatti, Entity Framework Core mette a disposizione altre funzionalità che possono tornare utili a seconda degli scenari.

Querying avanzato

Come abbiamo accennato in precedenza, il provider LINQ permette di utilizzare codice SQL scritto a mano e anche di mischiare codice SQL e LINQ in una unica query. Questo è reso possibile dal metodo `FromSql` della classe `DbSet<T>` il quale accetta in input una stringa SQL. Tramite questa stringa possiamo specificare di eseguire una funzione, una stored procedure oppure la clausola from di un comando SELECT, al quale poi possiamo agganciare i metodi di LINQ per formare un'unica query.

Esempio 10.19

```
// Esegue una stored procedure per recuperare ordini
var orders = context.Orders
     .FromSql("EXECUTE dbo.GetOrdersByCustomer {0}", customerId)
     .ToList();

// Concatena una query su una Table-Valued Function con LINQ
var orders = context.Orders
     .FromSql("Select * from dbo.GetOrdersByCustomer {0}", customerId)
     .Include(o => o.OrderDetails)
     .Where(o => o.ShipAddress.City == "Rome")
     .OrderBy(o => o.OrderDate)
     .ToList();
```

Nel primo esempio eseguiamo una funzione che restituisce gli ordini, nel secondo caso facciamo una query sul risultato della funzione, specificando che vogliamo estrarre anche i dettagli, che vogliamo prendere solo gli ordini destinati a `Roma` ordinati per data. Questa query viene interamente eseguita sul server grazie al fatto che il provider LINQ riesce a generare un'unica query.

Un'altra funzionalità importante è quella che prevede l'esecuzione delle query asincrone. Entity Framework Core introduce i metodi `ToListAsync`, `ToArrayAsync`, `FirstAsync`, solo per nominarne alcuni, su cui deve essere fatta l'`await` per aspettarne la fine dell'esecuzione. Va tuttavia specificato

che il contesto non è thread safe, quindi l'asincronia non può essere sfruttata per eseguire più query in parallelo.

L'ultima funzionalità da tenere a mente è la capacità di esprimere dei filtri a livello globale per gli entityset. Durante la fase di mapping possiamo specificare un filtro su una classe dell'object model ed Entity Framework Core applicherà sempre questo filtro. Questa tecnica torna utile quando abbiamo un database multi-tenant e vogliamo estrarre solo i record associati al tenant dell'utente loggato o quando usiamo la cancellazione logica dei record e vogliamo estrarre solo quelli non cancellati. In quest'ultimo caso, applicare un filtro globale che esclude quelli non cancellati ci consente di non dover scrivere il filtro in ogni query.

Esempio 10.20

```
//mapping con filtro globale
modelBuilder.Customers<Orders>().HasQueryFilter(c => !c.Deleted);

//query
var orders = await context.Customers.ToListAsync();
```

Nei casi in cui non vogliamo che i filtri siano onorati in una specifica query, dobbiamo concatenare alla query la chiamata al metodo `IgnoreQueryFilters`.

Come si è visto nel corso della sezione, la scrittura di query in Entity Framework Core è estremamente potente, nonostante i problemi di gioventù. Passiamo ora a vedere, invece, come persistere i dati sul database.

Salvare i dati sul database

Quando un oggetto viene recuperato dal database, viene anche memorizzato all'interno del contesto, in un componente chiamato **change tracker**.

Sono due i motivi per cui questo comportamento è necessario. Innanzitutto, ogni volta che eseguiamo una query, il contesto (che è responsabile della creazione fisica degli oggetti) verifica che non esista già un oggetto corrispondente (ovvero con la stessa chiave primaria) nel change tracker. In caso affermativo, il record proveniente dal database viene

scartato e il relativo oggetto nel change tracker viene restituito. In caso negativo, viene creato il nuovo oggetto, messo nel change tracker e, infine, restituito all'applicazione. Questo garantisce che vi sia un solo oggetto a rappresentare lo stesso dato sul database. Inoltre, il contesto deve mantenere traccia di tutte le modifiche fatte agli oggetti, così che poi queste possano essere riportate sul database. Il fatto di avere gli oggetti in memoria semplifica questo compito. Salvare i dati sul database significa persistere la cancellazione, l'inserimento e le modifiche delle entità. Questo processo viene scatenato attraverso la chiamata al metodo `SaveChanges` (o la sua controparte asincrona `SaveChangesAsync`) del contesto. Questo metodo non fa altro che scorrere gli oggetti modificati memorizzati nel contesto, iniziare una transazione, costruire ed eseguire il codice SQL per la persistenza e, infine, eseguire il commit o il rollback a seconda che vada tutto bene oppure no.

> *Se abbiamo più chiamate al metodo SaveChanges e vogliamo che queste siano eseguite in un'unica transazione, dobbiamo usare la classe TransactionScope. Poiché il supporto per TransactionScope è stato introdotto in .NET Core 2.1 e poiché va abilitato nei provider specifici per database, è possibile che alcuni provider inizialmente non lo supportino. È tuttavia probabile che in tempi brevi tutti i provider supportino queste API. Per sicurezza è bene consultare la documentazione del produttore.*

Per poter generare il codice SQL corretto, il contesto deve conoscere lo stato di ogni singolo oggetto quando effettua il salvataggio. Un oggetto può essere in quattro stati:

- `Unchanged`: l'oggetto non è stato modificato. Il processo di persistenza non scatena alcun comando per gli oggetti in questo stato.

- `Modified`: qualche proprietà semplice dell'oggetto è stata modificata. Il processo di persistenza genera un comando SQL di UPDATE per ogni oggetto in questo stato, includendo solo le colonne modificate.

- `Added`: l'oggetto è stato marcato per l'aggiunta sul database. Il processo di persistenza genera un comando SQL di INSERT per ogni oggetto in questo stato.

- `Deleted`: l'oggetto è stato marcato per la cancellazione dal database. Il processo di persistenza genera un comando SQL di DELETE per ogni oggetto in questo stato.

Quando un oggetto viene creato da una query, il suo stato è `Unchanged`. Nel momento in cui andiamo a modificare una proprietà semplice o una complessa, lo stato passa automaticamente a `Modified`. `Added` e `Deleted` si differenziano dagli stati precedenti, in quanto devono essere impostati manualmente. Per settare lo stato di un oggetto su `Added`, dobbiamo utilizzare il metodo `Add` della classe `DbSet<T>`, mentre per impostare un oggetto su `Deleted`, dobbiamo invocare `Remove`. Analizziamo ora queste casistiche in dettaglio.

Persistere un nuovo oggetto

In ogni applicazione che gestisce il ciclo completo di un'entità, il primo passo è l'inserimento. Come detto poco sopra, per persistere un nuovo oggetto basta utilizzare il metodo `Add` della classe `DbSet<T>`, passando in input l'oggetto da persistere. L'effetto è quello di aggiungere l'oggetto al contesto nello stato di `Added`.

Esempio 10.21

```
using (var ctx = new NorthwindEntities())
{
  var c = new Customer { /*Imposta proprietà customer*/ };
  ctx.Customers.Add(c);
  ctx.SaveChanges();
}
```

Come possiamo notare nell'esempio, inserire un nuovo cliente è estremamente semplice. Quando si deve inserire un ordine, invece, la situazione cambia leggermente, poiché un ordine contiene riferimenti ai dettagli e anche un riferimento al cliente.

In tal caso, il metodo `Add` si comporta in questo modo: imposta l'oggetto passato in input in stato di `Added`, poi scorre tutte le sue navigation property, così da aggiungere gli oggetti collegati al contesto, mettendoli nello stato di `Added`. In questo modo, possiamo richiamare una volta soltanto il metodo `Add`, passando l'ordine, con il vantaggio che tanto i

dettagli quanto il cliente verranno aggiunti al contesto in fase di inserimento.

Tuttavia, se per i dettagli questo rappresenta l'approccio corretto, per il cliente non possiamo dire altrettanto, visto che, in realtà, non deve essere inserito nuovamente. In questi casi dobbiamo ricorrere ai metodi `Attach` e `Update`. Entrambi i metodi, come `Add`, prendono un oggetto in input e ne scorrono tutte le navigation property per attaccarle al contesto, ma cambiano il modo di impostare lo stato. Per ogni oggetto, se il valore della chiave primaria è uguale al valore di default del suo tipo (0 per interi, null per stringhe e così via), allora il relativo stato viene impostato su `Added`, altrimenti viene impostato su `Unchanged` o `Modified`, a seconda del metodo usato.

Esempio 10.22

```
using (var ctx = new NorthwindEntities())
{
  var o = new Order {
    Customer = new Customer { CustomerId = "ALFKI" },
    //Imposta altre proprietà ordine ma non la chiave perché ordine nuovo
  };
  o.OrderDetails.Add(new OrderDetail {
    //Imposta proprietà dettaglio ma non la chiave perché ordine nuovo
  });
  o.OrderDetails.Add(new OrderDetail {
    //Imposta proprietà dettaglio ma non la chiave perché ordine nuovo
  });
  ctx.Orders.Attach(o);
  ctx.SaveChanges();
}
```

Nel caso dell'esempio 10.22, poiché non impostiamo la chiave né dell'ordine né dei dettagli, questi vengono aggiunti al contesto in stato di `Added`, mentre il cliente viene aggiunto al contesto in stato di `Unchanged`, visto che la sua chiave è impostata.

Ora che abbiamo imparato a inserire nuovi oggetti, passiamo ad analizzarne la rispettiva modifica.

Persistere le modifiche a un oggetto

Per modificare un cliente, basta leggerlo dal database, modificarne le proprietà e invocare il metodo `SaveChanges/SaveChangesAsync` come

mostrato nel prossimo esempio.

Esempio 10.23

```
using (var ctx = new NorthwindEntities())
{
  var cust = ctx.Customers.Find("ALFKI");
  cust.Address.Address = "Piazza del popolo 1";
  ctx.SaveChanges();
}
```

Questa modalità di aggiornamento viene definita come connessa, poiché l'oggetto è modificato mentre il contesto che lo ha istanziato è ancora in vita.

Tuttavia non è sempre così. Prendiamo, per esempio, un Web Service che metta a disposizione un metodo che torna i dati di un cliente e un altro che accetti un cliente e lo salvi sul database. La cosa corretta, in questi casi, è creare un contesto diverso in ogni metodo. Questo significa che non c'è nessun tracciamento delle modifiche fatte sul client e quindi il secondo contesto non può sapere cosa è cambiato. Quando incontriamo questo tipo di problema, si parla di modalità disconnessa, in quanto le modifiche all'oggetto sono fatte fuori dal contesto che lo ha creato.

In questi casi abbiamo a disposizione due modalità per risolvere il problema. Con la prima effettuiamo nuovamente la query per recuperare il cliente e impostiamo le proprietà con i dati che vengono passati in input al metodo. In questo caso il contesto riesce a tracciare le modifiche, quindi il codice è identico a quello visto nell'Esempio 10.24.

La seconda consiste nell'*attaccare* l'entità modificata al contesto, marcandola come `Modified` tramite il metodo `Update` già analizzato in precedenza. L'Esempio 10.24 mostra come usare questa tecnica.

Esempio 10.24

```
//Metodo che del servizio torna il cliente
using (var ctx = new NorthwindEntities())
{
  return ctx.Customers.Find("STEMO");
}
//Il client aggiorna il cliente e chiama il servizio di aggiornamento
Cust.Address.Address = "Piazza Venezia 10";
UpdateCustomer(Cust);
```

```
//Il servizio aggiorna il cliente
using (var ctx = new NorthwindEntities())
{
  ctx.Customers.Update(modifiedCustomer);
  ctx.SaveChanges();
}
```

Quando si parla di ordini e dettagli o, più in generale, di classi che hanno navigation properties, è importante sottolineare una cosa. Se aggiungiamo, modifichiamo o cancelliamo un dettaglio in modalità disconnessa, anche impostando l'ordine come modificato non otteniamo l'aggiornamento dei dettagli. Per eseguire un corretto aggiornamento, dobbiamo fare una comparazione manuale tra gli oggetti presenti sul database e quelli presenti nell'oggetto modificato e cambiare manualmente lo stato di ogni dettaglio. Dopo la modifica, è arrivato il momento di passare alla cancellazione dei dati.

Cancellare un oggetto dal database

L'ultima operazione di aggiornamento sul database è la cancellazione. Cancellare un oggetto è estremamente semplice, in quanto basta invocare il metodo `Remove` della classe `DbSet<T>`, passando in input l'oggetto. C'è comunque una particolarità molto importante da tenere presente. L'oggetto da cancellare **deve** essere attaccato al contesto. Questo significa che, anche in caso di cancellazione, abbiamo una modalità connessa e una disconnessa.

Nella modalità connessa possiamo semplicemente eseguire una query per recuperare l'oggetto e invocare poi `Remove` (Esempio 10.25).

Esempio 10.25

```
using (var ctx = new NorthwindEntities())
{
  var cust = ctx.Customers.Find("STEMO");
  ctx.Customers.Remove(cust);
}
```

Anche nella modalità disconnessa abbiamo due tipi di scelta: recuperare l'oggetto dal database e invocare `Remove` (stesso codice dell'esempio qui

sopra) oppure attaccare l'oggetto al contesto e poi chiamare il metodo `Remove` come nel prossimo esempio.

Esempio 10.26

```
using (var ctx = new NorthwindEntities())
{
  ctx.Customers.Attach(custToDelete);
  ctx.Customers.Remove(custToDelete);
}
```

Come abbiamo detto, gli oggetti attaccati al contesto vengono salvati nel change tracker. Questo componente espone alcuni metodi che possono essere utili per manipolare gli oggetti al suo interno.

Manipolare gli oggetti nel change tracker

Il contesto espone il change tracker tramite la proprietà `ChangeTracker` di tipo `ChangeTracker`. Il metodo più importante di questa classe è `Entries`, che ritorna una lista di oggetti di tipo `EntityEntry`. Ogni istanza di `EntityEntry` (detta anche entry) contiene i dati di un oggetto attaccato al contesto, li espone attraverso le sue proprietà e permette di manipolarli attraverso i suoi metodi. Quelli principali sono esposti nella Tabella 10.4.

Tabella 10.4 – Membri di `EntityEntry`.

Membro	Scopo
`Entity`	Proprietà che restituisce l'entity associata all'entry
`State`	Proprietà che restituisce e imposta lo stato dell'entity
`IsKeySet`	Specifica se la chiave dell'entity è impostata
`Metadata`	Restituisce i dati di mapping dell'entity
`OriginalValues`	Restituisce i valori dell'entity com'era quando è stata attaccata al contesto (recuperandola dal database o usando uno dei metodi per attaccarla al contesto)
`CurrentValues`	Restituisce i valori attuali dell'entity
`Reload/ReloadAsync`	Ricarica l'entity prendendo i valori dal database

Come abbiamo detto, `Entries` restituisce una lista di entry. Questo significa che possiamo filtrare le entry per recuperarne una specifica e modificarne lo stato, oppure tutte quelle in uno stato di `Added` per loggare le operazioni di inserimento o altro ancora.

Quando abbiamo un riferimento a una entity e vogliamo recuperare l'entry associata, possiamo usare il metodo `Entry` del contesto che accetta in input la entity e restituisce l'entry.

Aggiornare i dati con Entity Framework Core è tutt'altro che complesso, quindi non necessita ulteriori approfondimenti. Nella prossima sezione, invece, vedremo brevemente altre caratteristiche che possono tornare utili durante lo sviluppo.

Funzionalità aggiuntive di Entity Framework Core

Ovviamente, in questo capitolo abbiamo trattato solo gli argomenti principali che ci permettono di sviluppare con Entity Framework Core. Infatti, esistono molte altre funzionalità che questo framework ci mette a disposizione. In questa sezione elenchiamo quelle più comuni:

- Concorrenza: Entity Framework Core gestisce la concorrenza ottimistica. L'unica cosa che dobbiamo fare per abilitare la concorrenza è indicare nel mapping quali campi devono essere controllati.

- Code-First migration: permette di creare il database a design-time, partendo dalle classi del modello a oggetti. Inoltre, permette di modificare il database al cambio del codice delle classi mappate.

- Validazione: poiché nel mapping specifichiamo molte informazioni, come per esempio la lunghezza di una proprietà di tipo `String`, il contesto prima di persistere le notifiche verifica che le proprietà siano conformi al loro mapping, evitando così di arrivare al database con dati non validi.

- Logging: Entity Framework Core ha un motore di logging integrato, dove possiamo intercettare l'esecuzione dei comandi e generare un log.

- Mapping avanzato: Entity Framework Core permette di mappare i campi di una tabella su più classi, di mappare l'ereditarietà secondo il modello TPH, di mappare campi della tabella verso membri privati di una classe e di specificare chiavi univoche alternative.

Conclusioni

In questo capitolo abbiamo parlato delle caratteristiche principali di Entity Framework Core, così da poter cominciare a utilizzare questo O/RM. Ora siamo in grado di costruire e modificare un modello sia scrivendo a mano l'object model, il contesto e il mapping, sia sfruttando i tool che Entity Framework Core ci mette a disposizione per generare le stesse cose partendo dal database. Abbiamo visto come recuperare i dati dal database sfruttando il provider LINQ e come persistere sia oggetti semplici sia grafi complessi con poche righe di codice.

Tuttavia c'è molto di più. Come abbiamo visto nell'ultima sezione del capitolo, Entity Framework Core offre una serie di funzionalità che lo rendono uno strumento valido per l'accesso ai dati. Questo, unito alla capacità di girare anche su piattaforme Linux e MacOS, apre scenari finora impossibili da immaginare, il che fa capire come Microsoft stia investendo molto su questa tecnologia, che raffigura il presente e il futuro dell'accesso ai dati.

Tuttavia, Entity Framework Core è ancora molto giovane e necessita miglioramenti di stabilità, performance e funzionalità. In alcuni scenari, queste mancanze non sono un problema, mentre in altri rappresentano un ostacolo insormontabile. Prima di scegliere Entity Framework Core è sempre bene ponderare la scelta in base alle proprie esigenze.

Ora che abbiamo visto come accedere ai dati residenti su un database, nel prossimo capitolo vedremo come pubblicarli attraverso servizi REST.

11

Sviluppare servizi RESTful con ASP.NET Core WebAPI

Nel corso del libro abbiamo introdotto le caratteristiche del pattern MVC supportato da ASP.NET Core, parlando di Controller, Dependency Injection, routing e view. Nel capitolo precedente, invece, sono stati introdotti concetti molto legati ai dati per spiegare com'è possibile recuperarli ed elaborarli utilizzando O/RM come Entity Framework Core.

Date le basi già affrontate, all'interno di questo capitolo tratteremo una sorta di variazione sul tema: non è detto che i dati debbano sempre essere messi in comunicazione e mostrati all'interno di una view ma, al contrario, spesso vanno esposti, per realizzare servizi, che potranno anche essere riutilizzati, anche se magari solo in parte, nelle view di MVC.

Proprio per questi motivi, all'interno di questo capitolo introdurremo i concetti legati a WebAPI e REST API, ne capiremo le differenze e cercheremo di evidenziare i vantaggi nell'utilizzare sistemi di self-discovery per riuscire a sviluppare in modi separati client e server.

I servizi RESTful

REST (REpresentational State Transfer), al contrario di quello che si potrebbe credere, non impone una definizione di come deve essere costruita una API ma indica il modo in cui gli endpoint si comportano in funzione dell'URI specificato: l'utilizzo di JSON o XML come formato di risposta non è definito, perché REST non si occupa di protocolli. A voler

essere precisi, il protocollo HTTP non fa parte di REST, sebbene sia complesso immaginarsi scenari di API che non fanno uso di HTTP.

> *REST non è uno standard, ma uno stile architetturale con cui disegnare degli endpoint, in modo che sia più facile per tutti capire come invocarli.*

La definizione di REST passa attraverso questi principi fondamentali:

- Client (il consumatore del servizio) e server (il produttore) sono architetturalmente separati e devono evolvere in modo indipendente: non è necessario che il client si preoccupi di capire come vengono recuperati i dati lato server, così come per il server non è necessario capire come verranno utilizzati i dati nel client.

- Il servizio deve essere stateless: lo stato viene gestito richiesta per richiesta ed è per questo che viene garantita la scalabilità.

- I dati devono essere mantenibili: ogni risposta mandata dal server deve specificare la durata di validità dei dati (cache) e deve possibilmente essere immediata, ovvero composta architetturalmente da meno strati possibili (uno, nel caso ideale).

- Ogni risorsa deve essere identificata in modo univoco: le risorse devono essere separate dalla loro rappresentazione (XML o JSON che sia).

- Le risposte devono contenere informazioni necessarie per fare il discovery e spiegare al client come utilizzare le API (HATEOAS).

La maggior parte delle API di cui facciamo uso tutti i giorni, probabilmente senza accorgercene, non fa uso corretto di REST, ma questo non implica che non siano API valide o che non funzionino nel modo corretto. Sebbene sia un pattern architetturale, seguire i principi alla base di REST implica applicare dei suggerimenti a livello di design che non tutti gli sviluppatori e gli architetti sono disposti ad accettare, oppure sono in grado di seguire. Per individuare il grado di aderenza a

REST, esiste un modello, creato da Leonard Richardson e visibile più nel dettaglio su http://aspit.co/bnp, che spiega nel dettaglio che cosa può essere considerato davvero REST e che dà per scontato l'uso di HTTP come protocollo di comunicazione:

- Livello 0: viene utilizzato lo stesso endpoint (per esempio www.miosito.com/api) per fare tutte le chiamate ai vari servizi esposti, poiché gestiti internamente dall'endpoint. Diventa molto difficile per il client capire che cosa succede e fare il discovery di altre risorse oppure, più semplicemente, capire se deve aspettarsi dei dati in risposta, visto che viene utilizzato sempre lo stesso verb HTTP.

- Livello 1: viene specificato un endpoint differente per ogni risorsa, per esempio /api/ customers per lavorare con oggetti di tipo Customers, oppure /api/customers/1 per recuperare uno specifico customer. Ancora una volta non si fa uso dei verb dell'HTTP e quindi, tecnicamente, non si sta aderendo ad alcuno standard.

- Livello 2: ogni link viene invocato con uno specifico verb, per avere una determinata funzionalità: useremo, quindi, GET per leggere i dati, POST per fare inserimenti e così via. Inoltre, ogni risposta inviata al client deve contenere il corretto status code che identifica l'esito della chiamata (per esempio 200 (OK) per una chiamata andata a buon fine, 201 (Created) per una chiamata andata a buon fine e con un oggetto creato in POST).

- Livello 3: indica il supporto per HATEOAS e in ogni risposta deve essere specificato come raggiungere altri luoghi se necessario (per esempio, dopo la creazione di un Customer, sarebbe opportuno avere una location per poterlo raggiungere e visualizzare).

In tutte queste specifiche viene solamente menzionato il fatto di avere endpoint differenti per ogni chiamata, ma non viene definita in alcun

modo una convenzione dei nomi, pertanto viene rimandato tutto al buon senso e alla capacità del designer delle API.

Vengono comunque utilizzate, in genere, diverse pratiche, tra cui:

- L'uso di sostantivi al posto delle azioni specifiche negli URL: una chiamata a `/api/getcustomers` viene considerata sbagliata e dovrebbe essere sostituita con una chiamata `GET` ad `/api/customers`.

- Predicibilità: per recuperare un oggetto specifico, l'URL deve essere piuttosto immediato, quindi per recuperare un oggetto customer, viene considerato opportuno costruire un URL come `/api/customers/id` piuttosto che `/api/id/customers` perché l'id all'inizio dell'URL potrebbe essere conteso fra più risorse (libri, prodotti, clienti ecc.).

- L'uso di filtri in query string: tutto ciò che non è definibile come risorsa, quindi filtri e ordinamenti, dovrebbe fare parte di parametri della query string, non del path e, pertanto, un ordinamento potrebbe essere fatto come `/api/customers?orderby=firstName` anziché con `/api/customers/orderby/firstName`.

Nonostante l'introduzione di tutte queste regole, è importante riconoscere che l'uso di ASP.NET Core, come vedremo all'interno del capitolo, ma anche di una qualsiasi altra tecnologia, non permette la creazione di API "certificate" RESTful out-of-the-box: essendo un toolkit flessibile, fornisce tutti gli strumenti necessari per poterle realizzare con facilità. Resta compito degli architetti software progettare un sistema che sia in grado di lavorare secondo i principi definiti in precedenza.

Dando uno sguardo diretto ad ASP.NET Core e all'implementazione delle WebAPI, notiamo già una grossa differenza rispetto al passato: non ci sono più due framework separati, poiché la parte di ASP.NET MVC e quella di ASP.NET WebAPI sono integrate all'interno di un unico framework. ASP. NET Core MVC, infatti, è pensato e realizzato come un unico framework in grado di realizzare endpoint, a prescindere dalla tipologia di risposta che questi avranno.

Implementare un endpoint per le API con ASP.NET Core

La Figura 11.1 mostra l'infrastruttura condivisa tra ASP.NET Core MVC e ASP.NET Core Web API. Possiamo notare come, di fatto, il controller sia uno, a prescindere dal tipo di ritorno, e la differenza lo faccia il modo in cui gli sviluppatori vanno a scrivere effettivamente la action.

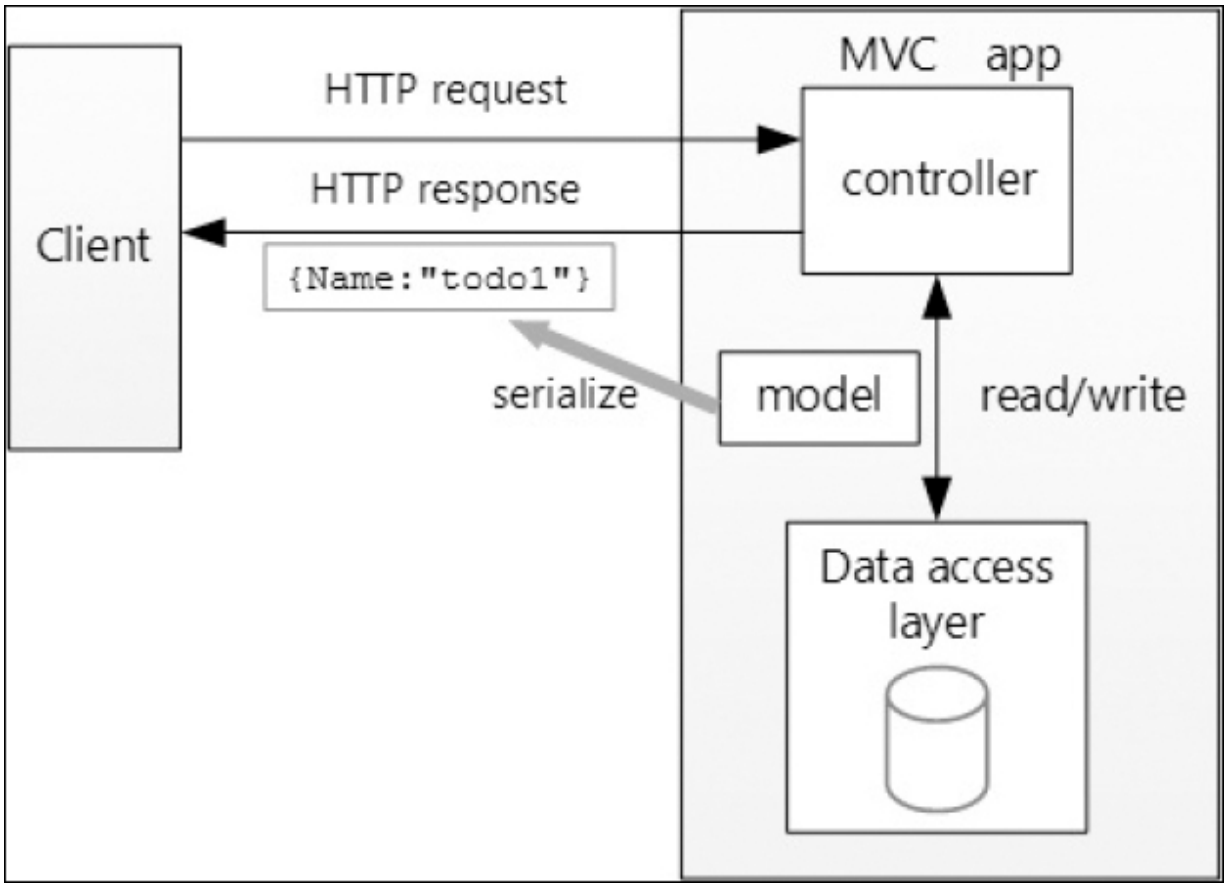


Figura 11.1 – Architettura di una ASP.NET Core WebAPI classica.

Il flusso è semplice: il client manda la richiesta HTTP verso il server (che possiamo simulare con un tool come Postman o curl), il server intercetta la chiamata tramite un controller e le relative action, passando tramite diversi layer di servizi e, eventualmente, interrogando il database, per poi ritornare la risposta tramite un modello serializzato come risposta.

Nell'esempio che vedremo illustrato di seguito, non andremo a fare uso di Entity Framework Core e di un database vero e proprio, perché non è strettamente necessario per illustrare il funzionamento delle API: tutti i dati generati ed elaborati saranno fittizi ma comunque utili per spiegarne il flusso. In particolare, è già stata preparata la classe `Customer`, che identifica un set di clienti che possono acquistare una serie di prodotti di tipo `Product`, mentre tutte le operazioni tipiche, ovvero l'aggiunta, la rimozione e la lettura, passano attraverso una classe chiamata `CustomerRepository` (e la relativa interfaccia `ICustomerRepository`) che viene iniettata all'interno del costruttore come servizio transient.

Il codice è disponibile nell'Esempio 11.1.

Esempio 11.1

```
public class CustomersController : Controller
 {
  private readonly ICustomerRepository customerRepository;
  public CustomersController(ICustomerRepository customerRepository)
  {
    this.customerRepository = customerRepository;
  }
 }
```

Dal precedente codice possiamo già notare come il controller appena creato erediti a tutti gli effetti dalla classe `Controller`, al contrario di quanto avveniva con ASP.NET WebAPI, in cui i controller ereditavano da `WebApiController`, rendendo i controller di ASP.NET MVC incompatibili con quelli di ASP.NET Web API.

Aggiungere la prima API che va a leggere e ritornare l'elenco di tutti i clienti è facile tanto quanto scrivere un metodo, come possiamo vedere nell'Esempio 11.2.

Esempio 11.2

```
public JsonResult GetCustomers()
{
  var customers = customerRepository.GetCustomers();
  // restituiamo un risultato di tipo JSON
```

```
    return new JsonResult(customers);
  }
```

In questo caso specifico, viene richiesto al repository che si occupa della gestione dei clienti di ritornare l'elenco degli stessi e, quindi, viene fatta una serializzazione dei dati in oggetti con il formato JSON, attraverso la classe `JsonResult`, che rimanda tutti i dati al client che ha effettuato la richiesta. Se proviamo ad avviare il server e a eseguire la richiesta con Postman, noteremo però un errore 404 in risposta alla chiamata su `/api/customers`, come viene illustrato nella Figura 11.2.

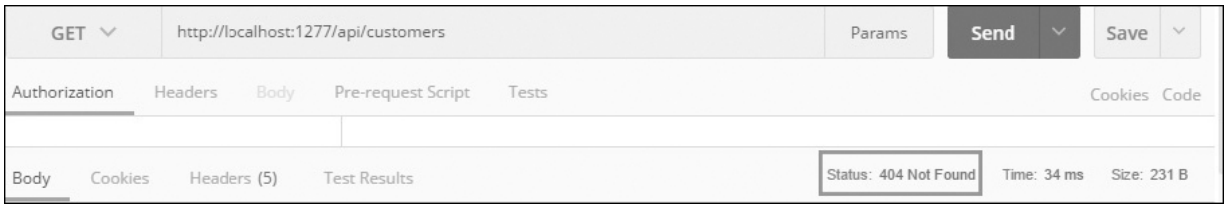


Figura 11.2 – La chiamata `GET` verso un endpoint non configurato produce un errore.

Questo errore è del tutto normale, perché non è stato ancora comunicato ad ASP.NET Core che vogliamo che l'endpoint risponda su un determinato URL.

Per farlo, è necessario fare uso del routing, così da mappare la action del controller a un URL specifico: nonostante esistano due tipi di routing, convention-based e attribute-based, Microsoft consiglia di utilizzare il primo per la definizione delle rotte sulla parte "front-end" di MVC, ovvero quella che viene poi utilizzata dal sito web, mentre consiglia di fare uso degli attributi per lavorare con le API.

Per rispondere al link corretto che è stato predisposto in Postman, è necessario modificare gli Esempi 11.1 e 11.2 così che assumano una forma come quella mostrata nell'Esempio 11.3.

Esempio 11.3

```
  [Route("api/customers")]
```

```
public class CustomersController : Controller
{
  //...
  [HttpGet]
  public JsonResult GetCustomers()
  {
    var customers = customerRepository.GetCustomers();
    return new JsonResult(customers);
  }
}
```

Nell'Esempio 11.3 è stata registrata una route che risponde all'indirizzo `/api/customers` e, grazie all'uso dell'attributo `HttpGet`, la risposta sarà servita solo se il verb HTTP con la quale viene invocato l'endpoint è quello `GET`.

In realtà, possiamo fare uso delle convenzioni di ASP.NET Core per modificare il valore dell'attributo `Route` in api/[controller]: così facendo, il nome del controller viene passato in automatico all'attributo che gestisce la route ma, in caso di refactoring, tutte le API che vengono esposte dal controller corrente cambierebbero l'endpoint. Si tratta di una scelta personale, anche se di solito è fortemente sconsigliato lasciare l'endpoint dinamico, a meno che venga usato in concomitanza con un sistema di API versioning tipo Swagger, di cui vedremo i dettagli nel corso del prossimo capitolo.

Una volta configurata correttamente la route, il risultato ottenuto sarà simile a quello illustrato nella Figura 11.3.

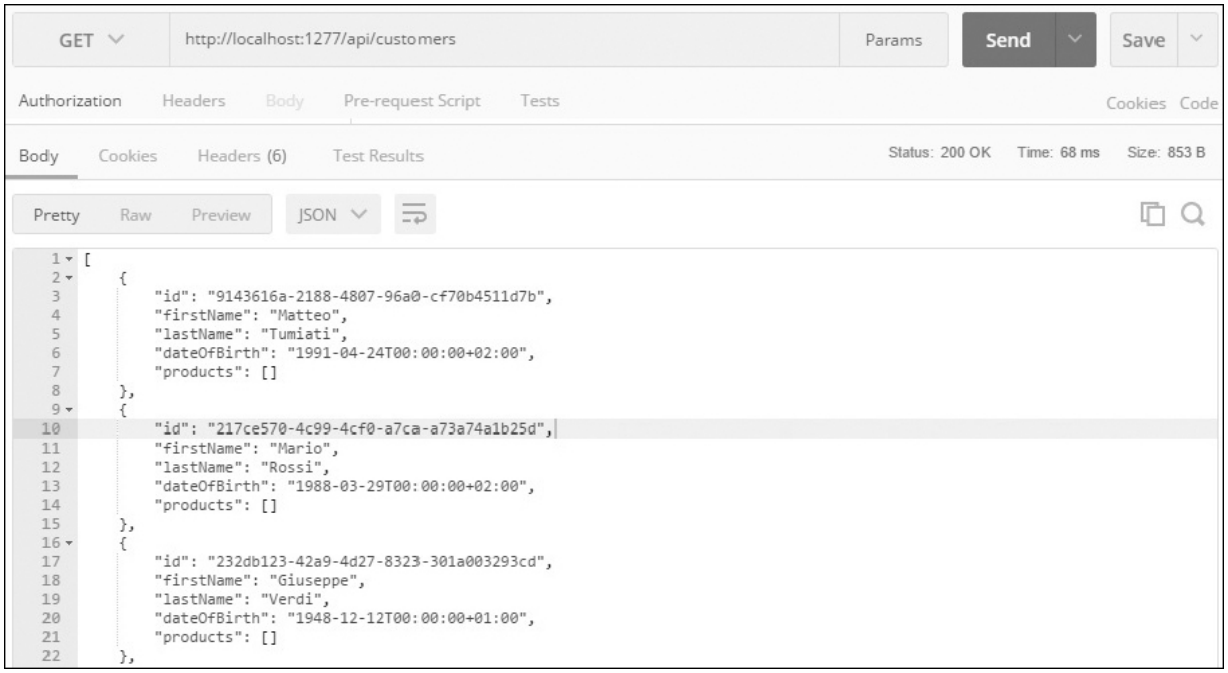


Figura 11.3 – Ecco come si presenta la risposta a una chiamata con il verb `GET` **con l'elenco di oggetti di tipo** `Customer`**.**

In base alla risposta ottenuta, possiamo osservare che, poiché la funzione sta mandando in risposta un oggetto di tipo JSON, viene mantenuta la notazione corretta delle proprietà in camelCase. Inoltre, anche se forse meno ovvio, stiamo mandando in risposta un oggetto identico a quello che è stato potenzialmente estratto a partire dal database e, pertanto, così facendo vengono esposte proprietà che potrebbero non essere di interesse pubblico o rientrare negli scopi per cui viene definita questa API: è importante, infatti, avere una separazione, non solo per proteggere il database da eventuali attacchi, ma anche perché, in caso di aggiornamenti della struttura dati, l'API pubblicata manterrebbe un risultato identico, senza il bisogno di agire sul versioning. Nell'esempio proposto, potrebbe essere utile, infatti, ritornare una sola proprietà con il nome completo, anziché esporre nome e cognome come proprietà separate, oppure potrebbe fare comodo mostrare direttamente l'età, anziché mostrare la data di nascita e lasciare il calcolo al client.

Nell'Esempio 11.4, pertanto, facciamo uso di un DTO (Data Transfer Object), cioè di un modello predisposto esattamente allo scopo di

trasferire dati all'interno di servizi.

Esempio 11.4

```
[HttpGet]
public JsonResult GetCustomers()
{
  var customers = customerRepository.GetCustomers();
  var customersForRead = customers.Select(customer => new CustomerForRead
  {
    Id = customer.Id,
    Name = $"{customer.FirstName} {customer.LastName}",
    Age = customer.DateOfBirth.GetAge()
  });
  return new JsonResult(customersForRead);
}
```

L'Esempio 11.4 dimostra come, tramite una semplice query di LINQ, sia possibile creare una nuova lista di oggetti di tipo `CustomerForRead`, in cui, al contrario della classe `Customer`, sono state esposte solo le proprietà da trasportare, come `Id`, utile per recuperare eventualmente l'utente singolo in un momento successivo, `Name` che espone il nome completo e `Age` che rappresenta l'età a partire dalla data di nascita grazie all'uso di un extension method costruito ad-hoc per questo esempio. L'esempio in questione funziona bene nel caso in cui ci siano da modificare poche proprietà ma, in caso di oggetti complessi e di API che espongano più volte lo stesso modello, potrebbe diventare noioso riscrivere più e più volte lo stesso codice, e, inoltre, si correrebbe il rischio di commettere errori.

Per nostra fortuna, ci sono utility come AutoMapper, una libreria disponibile su NuGet che si occupa di object-mapping e cioè, tramite l'utilizzo di convenzioni, è in grado di convertire un oggetto in ingresso in un oggetto in uscita che ha proprietà simili, oppure proprietà calcolate secondo le direttive impostate a livello centrale. La libreria, come tutta la relativa documentazione, è open source e disponibile su GitHub all'indirizzo `http://aspit.co/bnq`.

Una volta installata la libreria, è necessario applicare un po' di configurazione per istruire il motore di mapping all'interno del metodo `Configure` della classe `Startup`. L'Esempio 11.5 mostra come fare.

Esempio 11.5

```
AutoMapper.Mapper.Initialize(config =>
{
  config.CreateMap<Customer, CustomerForRead>()
    .ForMember(o => o.Name, i => i.MapFrom(m => $"{m.FirstName} {m.LastName}"))
    .ForMember(o => o.Age, i => i.MapFrom(m => m.DateOfBirth.GetAge()));
});
```

A questo punto, poiché il mapping viene fatto in tutti i casi in cui dovesse essere necessario dalla libreria, registrata globalmente, nella funzione `GetCustomers` è possibile eliminare la query LINQ che si occupa della trasformazione dei dati, sostituendola con una chiamata al motore di mapping, come è visibile nell'Esempio 11.6.

Esempio 11.6

```
[HttpGet]
public JsonResult GetCustomers()
{
  var customers = customerRepository.GetCustomers();
  var customersForRead =
                Mapper.Map<IEnumerable<CustomerForRead>>(customers);
  return new JsonResult(customersForRead);
}
```

Il risultato ottenuto sarà simile a quello illustrato nella Figura 11.4.

```
1  [
2      {
3          "id": "ec3d723c-fca5-455a-bfb2-13be9e54caa1",
4          "name": "Matteo Tumiati",
5          "age": 27
6      },
7      {
8          "id": "1cd20b70-2028-41c7-b6b5-da7093c65245",
9          "name": "Mario Rossi",
10         "age": 30
11     },
12     {
13         "id": "5efc3f67-3d42-42b7-b4c1-ee4ef9e2a364",
14         "name": "Giuseppe Verdi",
15         "age": 69
16     },
17     {
18         "id": "bd627a3a-7ae4-4d5c-bb43-dd98394e75b5",
19         "name": "Marco Bianchi",
20         "age": 43
21     }
22 ]
```

Figura 11.4 – Manipolazione dei dati in risposta grazie al mapping con AutoMapper sulla classe DTO.

Poiché il mapping viene ora fatto verso la classe DTO `CustomerForRead`, come si nota nella Figura 11.4, i dati ottenuti in risposta non sono più i dati originali esposti dalla classe `Customer`, come per esempio la data di nascita, ma sono i dati manipolati da AutoMapper che va a esporre la sola età e il nome completo anziché i due valori come campi separati.

Quando si parla di recupero dei dati e di servizi internet, però, è sempre bene prestare attenzione al fatto che tutto può andare storto, dalla connessione internet mancata fra due servizi che si parlano, all'irraggiungibilità di un database, fino a un errore nel codice. Finora abbiamo dato per scontato che il servizio funzioni sempre e che produca una risposta corretta ma, nel caso in cui si verifichino degli errori, è bene avvisare il client così che, eventualmente, possa riprovare la chiamata per avere la risposta che si aspetta.

Status code e gestione degli errori

Una volta ottenuto l'elenco dei clienti, il prossimo passo è quello di realizzare un'API che esponga una singola entità. La costruzione di questa API è piuttosto semplice e la logica continua a dipendere dall'architettura scelta per il progetto, come possiamo notare nell'Esempio 11.7.

Esempio 11.7

```
[HttpGet("{id}")]
public JsonResult GetCustomer(Guid id)
{
  var customer = customerRepository.GetCustomer(id);
  var customerForRead = Mapper.Map<CustomerForRead>(customer);
  return new JsonResult(customerForRead);
}
```

Per ottenere una singola entità, viene usata quella che è la sua chiave primaria, in modo tale che l'oggetto venga identificato in modo univoco e, come viene dimostrato nell'Esempio 11.7, la firma del metodo accetta in ingresso un parametro di tipo `Guid`, il cui nome corrisponde a quello della chiave primaria `Id`. Il nome della proprietà deve essere mantenuto identico a quello esplicitato nella route dell'attributo `HttpGet`, in modo che il framework sia in grado di ricostruirlo al momento della richiesta: in caso non ci fosse una corrispondenza diretta, infatti, le proprietà del metodo assumeranno il loro valore di default, variabile a seconda del tipo: questo comportamento è del tutto simile a quanto abbiamo già visto nei capitoli precedenti ed è legato al funzionamento del routing.

La richiesta, in questo caso, verrà mappata sull'endpoint `/api/customers/{id}` e verrà restituito l'oggetto corrispondente all'ID selezionato. Qualora non ci sia un elemento corrispondente, a fronte di una richiesta valida, la risposta che si otterrà è forse più naturale ma meno funzionale del previsto e la possiamo vedere nella Figura 11.5.

Figura 11.5 – In caso di mancanza di dati, anziché una risposta errata, si ottiene comunque una risposta di tipo 200 (OK).

Osservando nel dettaglio, l'oggetto inviato in risposta è di tipo `CustomerForRead`, per cui viene inviata una risposta di tipo `null`, perché una corrispondenza nello storage non è stata trovata. Per questo motivo, lo status code inviato con la risposta è 200 (OK), che in questo caso è formalmente sbagliato, poiché la richiesta non è andata a buon fine.

Parte di un buon design di servizi REST è il fare uso del corretto status code HTTP. Questo è importante, perché i client che andranno a leggere e a elaborare i dati ottenuti dalle API esposte potranno capire in automatico se le richieste sono state eseguite correttamente oppure se si sono verificati errori e, in questo caso specifico, capirne la tipologia, per ritentare eventualmente la chiamata all'API corrispondente.

Gli status code fanno parte dello standard HTTP e si dividono principalmente in cinque macro-categorie:

- `100`: sono codici informativi aggiunti dopo l'introduzione dello standard HTTP;

- `200`: indicano che lo stato della richiesta è andato a buon fine; in particolare, 200 (OK) rappresenta una chiamata che è andata a buon fine, 201 (Created) è associato alla creazione di un nuovo elemento, mentre 204 (NoContent) viene utilizzato quando la risposta non ha un contenuto vero e proprio ma è stata completata con successo;

- `300`: sono utilizzati principalmente per indicare un redirect. Pertanto nelle API possiamo utilizzarli per rimandare il client al nuovo indirizzo. Con 301 (Moved Permanently) indichiamo che la risorsa ha un nuovo URL definitivo, mentre con 302 (Found) che la stessa è disponibile temporaneamente su un altro URL;

- `400`: indicano errori avvenuti nella chiamata. Per esempio, con 401 (Unauthorized) segnaliamo che la richiesta non ha i permessi necessari a essere eseguita, mentre con 404 (Not Found) si indica

una risorsa non trovata e con 405 (Method Not Allowed) che l'URL è stato richiamato con un verb HTTP non supportato;

- `500`: è una tipologia di errore lato server, in cui il client non può fare nulla, se non riprovare la chiamata successivamente.

Quelli elencati in precedenza sono solamente alcuni dei tanti codici che si possono trovare sviluppando le API o servizi web in genere, mentre altri li affronteremo nel corso di questo stesso capitolo. Pertanto, il codice dell'Esempio 11.7 può essere modificato per gestire l'esistenza di un cliente, così da gestire correttamente la risposta, come viene dimostrato nell'Esempio 11.8.

Esempio 11.8

```
[HttpGet("{id}")]
public IActionResult GetCustomer(Guid id)
{
  var customer = customerRepository.GetCustomer(id);
  if (customer == null)
    return NotFound();
  var customerForRead = Mapper.Map<CustomerForRead>(customer);
  return Ok(customerForRead);
}
```

Nell'Esempio 11.8 possiamo notare come ci siano diverse modifiche rispetto all'esempio mostrato in precedenza: la prima fra tutte è il tipo di ritorno, che non è più un `JsonResult` ma è stato reso generico sul tipo `IActionResult`: discuteremo dei vantaggi più avanti, nel corso di questo stesso capitolo.

Inoltre, abbiamo aggiunto un controllo sull'esistenza del nostro cliente: in caso in cui l'oggetto contenga il valore `null`, ovvero che non sia presente all'interno dello storage, viene rimandata la risposta a una funzione `NotFound`, che cambierà lo status code al suo corrispondente, ovvero 404 (Not Found), esattamente come ci si aspetta, senza ritornare alcun elemento o alcun valore `null`, come possiamo notare nella Figura 11.6.

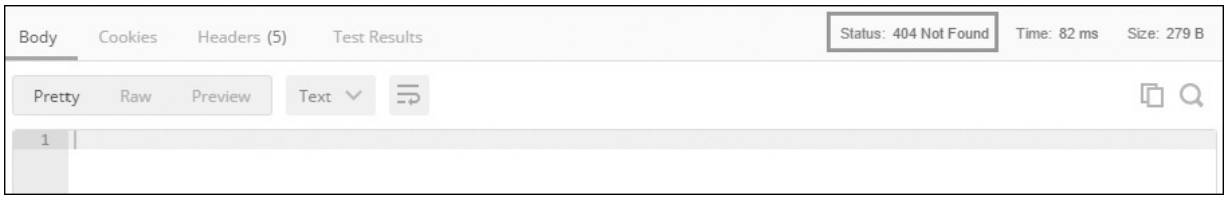


Figura 11.6 – Ecco come appare una risposta con status code 404 (Not Found), in caso di dati mancanti.

Nel caso in cui l'oggetto fosse presente, verrà restituito tramite la funzione `Ok`, che produrrà in risposta lo status code 200 (OK), come abbiamo già visto in precedenza. È interessante notare che queste due funzioni, così come altre già presenti in `Controller`, non fanno altro che wrappare una chiamata base al metodo `StatusCode`, in cui viene passato l'effettivo valore di stato. Un esempio simile potrebbe essere quello realizzato in caso di eliminazione di un elemento, in cui i valori di risposta possono differire, come viene mostrato nell'Esempio 11.9.

Esempio 11.9

```
[HttpDelete("{id}")]
public IActionResult DeleteCustomer(Guid id)
{
  if (!customerRepository.CustomerExists(id))
    return StatusCode(StatusCodes.Status404NotFound);
  customerRepository.DeleteCustomer(id);
  return StatusCode(StatusCodes.Status204NoContent);
}
```

Sebbene con questo approccio possiamo gestire i codici di stato per eventuali problemi nella chiamata effettuata lato client, non siamo ancora in grado di sfruttare a modo il codice 500 (Internal Server Error), che indica un errore del server. Questo errore, in genere, può capitare nel caso in cui ci sia un'eccezione non gestita: il client non può fare altro che tentare nuovamente la chiamata, nella speranza che il problema sia stato solo temporaneo. Una semplice chiamata che genera un'eccezione potrebbe essere quella esposta nell'Esempio 11.10.

Esempio 11.10

```
[HttpGet(″api/exception″)]
public IActionResult GenerateException()
{
  try
  {
    throw new NotImplementedException(″Questo metodo è finto.″);
  }
  catch (Exception)
  {
    return StatusCode(StatusCodes.Status500InternalServerError, ″Qualcosa è
andato storto.″);
  }
}
```

L'eccezione è stata gestita tramite un blocco `try…catch` e, poiché siamo in grado di capire cos'è andato storto, è meglio ritornare al client l'informazione generica di errore, senza esporre troppo le informazioni relative al problema, principalmente per motivi di sicurezza. Il risultato della chiamata sarà come quello mostrato nella Figura 11.7.

Figura 11.7 – Ecco come appare un errore 500 (Internal Server Error) in presenza di eccezioni gestite.

Purtroppo, il codice che viene scritto non è detto che venga testato e sia funzionante nella sua totalità, per questo possono esistere casi in cui qualche chiamata potrebbe ancora non essere gestita e generare errori di vario tipo. In questo caso, è bene fare uso del middleware `UseExceptionHandler` nel metodo `Configure` della classe `Startup`, per modificare l'eventuale risposta prima di rimandarla al client, come possiamo notare nell'Esempio 11.11.

Esempio 11.11

```
app.UseExceptionHandler(options =>
{
  options.Run(async context =>
  {
    context.Response.StatusCode = StatusCodes.Status500InternalServerError;
    await context.Response.WriteAsync("Qualcosa è andato storto.");
  });
});
```

A tutti gli effetti, il blocco `try…catch` dell'Esempio 11.10 diventa superfluo: tutte le richieste, a questo punto, verranno filtrate e gestite dal middleware, prima di arrivare al client, ottenendo, di fatto, lo stesso risultato di mettere un blocco `try…catch` direttamente dentro ciascuna action. Le possibili eccezioni non è detto che si verifichino solamente nel codice: infatti, è anche possibile che si verifichino problemi durante l'invio della chiamata dal client verso il server, soprattutto quando i due cercano di parlare "lingue" diverse. Per questo motivo, è necessario specificare a entrambi i meccanismi con cui devono comunicare.

Formatter e content negotiation

Quando si parla di WebAPI si dà spesso per scontato che il formato della risposta ottenuta dai servizi debba per forza di cose essere in JSON ma, come è già stato illustrato all'inizio di questo capitolo, JSON è solo il formato più diffuso e pertanto il formato di rappresentazione dei dati è piuttosto indifferente. Inoltre, ci sono sistemi che lavorano con altri formati, per esempio XML oppure, nel caso di device IoT (Internet Of Things), con formati personalizzati e talvolta proprietari.

ASP.NET Core è in grado di supportare tutti gli scenari, perché tutta la sua pipeline è composta da micro-servizi sostituibili: qualora fosse necessario specificare il formato in ingresso o in uscita dal servizio, ASP.NET Core sarebbe in grado di consentircelo. A un livello più basso, però, è opportuno specificare in una sorta di contratto come client e server devono comunicare, e questo viene solitamente fatto tramite le header HTTP, con una tecnica che viene definita content negotiation.

Se il client che invia la richiesta specifica (tramite l'header `Accept`) un valore corrispondente ad `application/xml`, vuole assicurarsi che la

risposta sia in formato XML. Il server, al contrario, può specificare quali formati è in grado di gestire e, eventualmente, rifiutare le chiamate con uno status code 406 (Not Acceptable), come illustrato nell'Esempio 11.12.

Esempio 11.12

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddMvc(options =>
  {
    options.ReturnHttpNotAcceptable = true;
  });
}
```

La trasformazione tra oggetti e contenuto formattato in JSON piuttosto che in XML o in altri formati personalizzati viene effettuata dal runtime, in automatico, al momento dell'invio della risposta verso il client tramite i `Formatters`. Un esempio di registrazione è disponibile nell'Esempio 11.13.

Esempio 11.13

```
services.AddMvc(options =>
{
  options.OutputFormatters.Add(new XmlDataContractSerializerOutputFormatter());
});
```

Nell'Esempio 11.13 è stato aggiunto il formatter per la codifica degli oggetti in formato XML, oltre a quelli di default già registrati, come JSON. Per costruirne uno personalizzato è necessario creare una classe che erediti da `TextOutputFormatter`, che implementi la logica prevista, e poi aggiungerlo alla lista degli `OutputFormatters`. Il formatter viene scelto in automatico dal framework in base al valore che è in grado di leggere dall'header di `Accept` durante l'esecuzione della richiesta ma, qualora questo non venga specificato, verrà ritornato il valore di default, ovvero il primo nell'elenco dei formatter. Supponendo di richiedere l'elenco dei

clienti in formato XML, l'output ottenuto dalla richiesta sarà simile a quello della Figura 11.8.

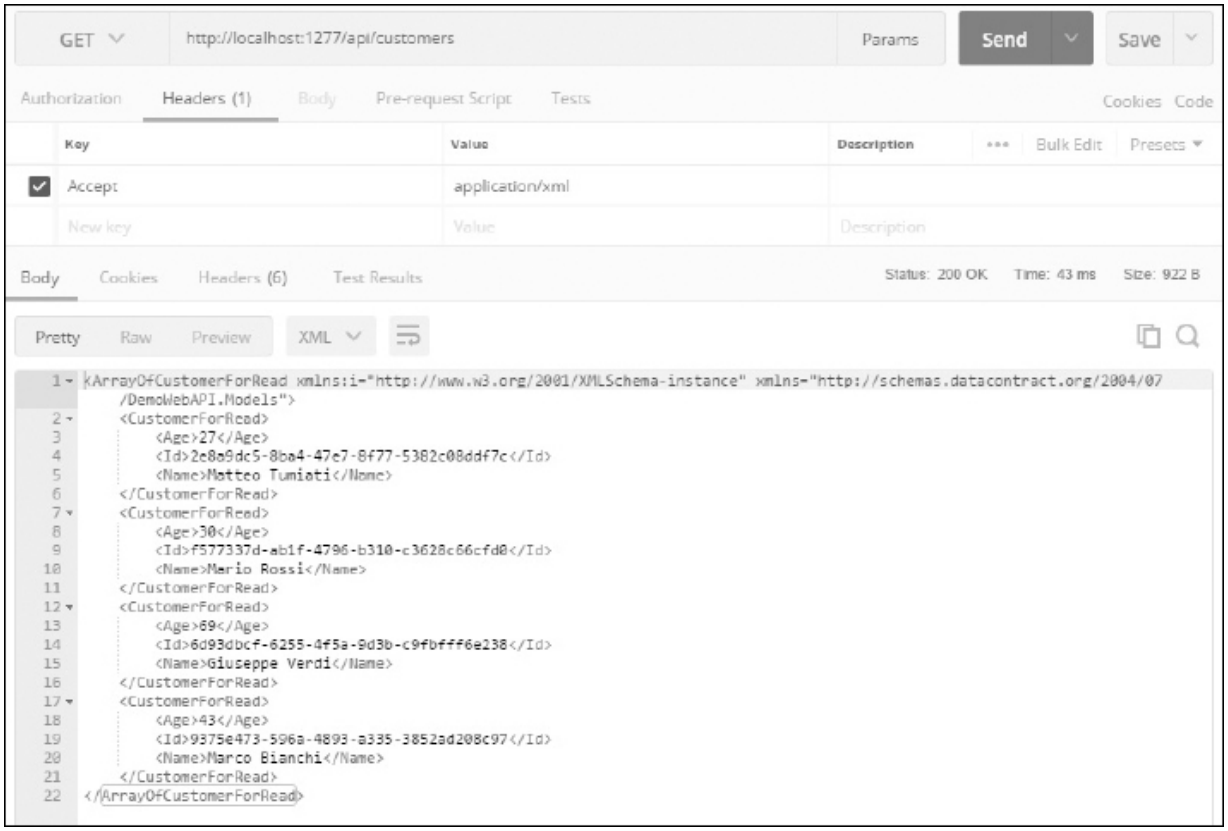


Figura 11.8 – Risposta con serializzazione in formato XML per via della richiesta con header di `Accept`.

Quello che non abbiamo ancora visto è la creazione di un'API che consenta di aggiungere un nuovo cliente. Quando si effettua la creazione, solitamente è bene utilizzare l'adeguato verb HTTP `POST` e caricare i dati che devono essere interpretati da ASP.NET Core nel body della richiesta, come è visibile nell'Esempio 11.14.

Esempio 11.14

```
[HttpPost]
public IActionResult CreateCustomer([FromBody] CustomerForCreate customer)
{
  if (customer == null || !ModelState.IsValid)
    return BadRequest();
```

```
  var mappedCustomer = Mapper.Map<Customer>(customer);
  customerRepository.AddCustomer(mappedCustomer);
  if (!customerRepository.SaveChanges())
    return StatusCode(StatusCodes.Status500InternalServerError, "Impossibile
salvare le modifiche.");
  var createdCustomer = Mapper.Map<CustomerForRead>(mappedCustomer);
    return CreatedAtAction(nameof(GetCustomer), new { id = createdCustomer.Id },
createdCustomer);
}
```

In questo caso dobbiamo notare l'attributo `FromBody`, che indica al model binder di recuperare e trasformare il contenuto del body della richiesta in un oggetto di tipo `CustomerForCreate`. Questa nuova classe viene utilizzata per separare completamente il DTO di lettura da quello di scrittura: in questo caso, non c'è più bisogno di proprietà come `Id`, dato che sono calcolate in fase di inserimento, oppure del nome completo, dato che i campi sul database saranno fisicamente separati in due colonne distinte. Una volta ottenuto l'oggetto ed entrati nella funzione, è necessario controllare che sia stato ricostruito correttamente e che tutte le proprietà siano valide.

Successivamente, tramite AutoMapper, andiamo a convertire l'oggetto da `CustomerForCreate` a `Customer`, in modo che possa essere persistito nella base dati. Qualora la funzione di salvataggio non andasse a buon fine, dobbiamo inviare uno status code 500 (Internal Server Error) per segnalare che l'operazione non è stata completata con successo, in modo che il client possa riprovare a eseguirla in futuro. Se, al contrario, il salvataggio si completa correttamente, è utile rimappare l'oggetto recuperato (che ora ha la proprietà `Id` configurata in modo opportuno) in un oggetto da restituire al client: così facendo, il client è in grado di capire che l'oggetto è stato effettivamente creato e lo può già utilizzare a partire dalla risposta ottenuta che, grazie alla funzione `CreatedAtAction`, ritornerà proprio uno status code di 201 (Created), come possiamo vedere nella Figura 11.9.

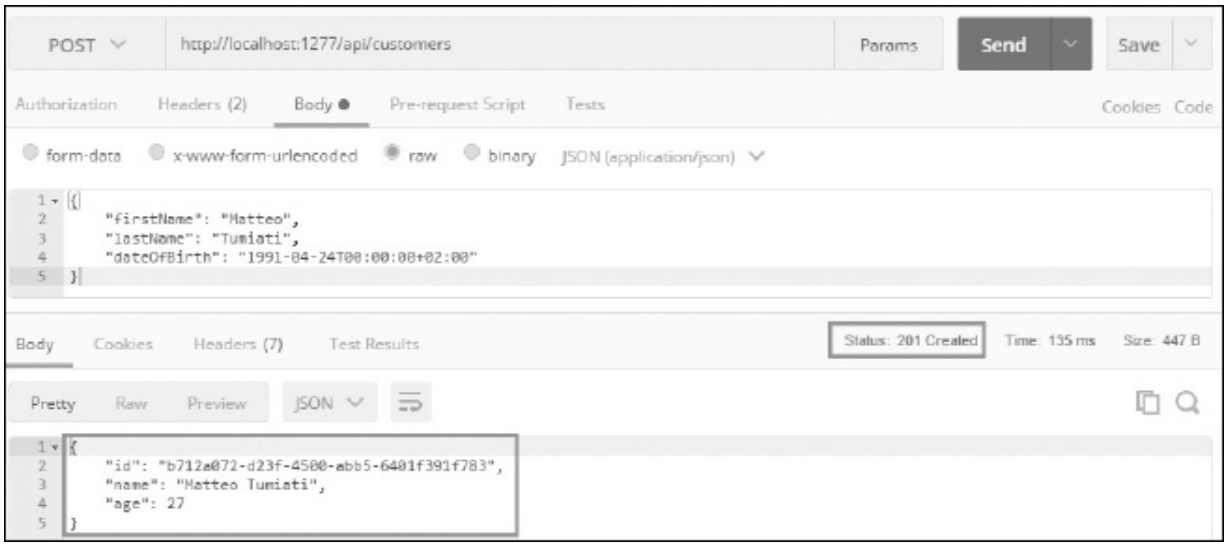


Figura 11.9 – Ecco come avviene una risposta 201 (Created) in caso di creazione di un nuovo oggetto `Customer`**.**

La risposta di tipo `CreatedAtAction` non ha restituito al client solo lo status code 201 (Created) e l'oggetto effettivamente creato, ma è anche andata a impostare l'header `Location` in modo che punti alla route che consente di recuperare quello stesso cliente appena creato.

Figura 11.10 – L'header `Location` **in risposta a uno status code 201 (Created) serve a rimandare il client al nuovo URL, da cui recuperare le informazioni circa l'oggetto appena creato.**

Tutta questa introduzione sull'inserimento di un nuovo elemento è utile per capire il concetto relativo all'header `Content-Type`: al contrario

dell'header `Accept`, che è in grado di fare content negotiation sulla risposta, l'header `Content-Type` è in grado di fare la negoziazione sul formato del contenuto inviato all'interno del body della richiesta. Nella Figura 11.9 possiamo notare come questa header venga impostata automaticamente da Postman durante la scrittura del body, una volta che è stato in grado di riconoscere il testo come JSON. Questa negoziazione, però, ha senso solamente se anche lato server esiste il provider in grado di deserializzare e convertire il body nel formato corretto: nel caso di `Accept`, poiché i dati dovevano essere mandati in output, si sono utilizzati gli `OutputFormatters`, mentre nel caso di `Content-Type`, poiché i dati devono essere manipolati in fase di input, si utilizzano gli `InputFormatters`. Nell'Esempio 11.15 registriamo entrambi.

Esempio 11.15

```
services.AddMvc(options =>
{
  options.ReturnHttpNotAcceptable = true;
  options.OutputFormatters.Add(new XmlDataContractSerializerOutputFormatter());
  options.InputFormatters.Add(new XmlDataContractSerializerInputFormatter());
});
```

Abbiamo configurato anche un convertitore dal formato XML per i dati in ingresso ma, allo stesso modo degli `OutputFormatters`, è relativamente immediato costruire un formatter personalizzato secondo le proprie esigenze e i propri protocolli. Poiché non è fra gli scopi del libro quello di entrare in scenari troppo avanzati e costruiti su misura, lasciamo al lettore un rimando alla documentazione ufficiale, necessaria per poter costruire un formatter custom. La potete trovare all'indirizzo: `http://aspit.co/bnm`.

Finora quello che è stato affrontato ha riguardato l'elaborazione di singoli elementi. Ma come bisogna comportarsi nei casi in cui ci siano grandi quantità di dati da spostare da un sistema a un altro?

Lavorare con le collection

Ci sono diversi casi, come quello del bulk insert, in cui si può verificare la necessità di caricare i dati da una collection. Per come è stato strutturato il routing, non c'è una modalità per caricare una lista di dati con una route univoca, infatti/api/customers è già stata utilizzata per lavorare con i singoli oggetti. La soluzione più immediata a questo problema è di creare una nuova risorsa, con un nuovo controller, specifico per lavorare con array di oggetti, come viene mostrato nell'Esempio 11.16.

Esempio 11.16

```
[HttpPost]
[Route(″/api/customerscollection″)]
public IActionResult CreateCustomerCollection([FromBody]
IEnumerable<CustomerForCreate> customers)
{
  if (customers == null || !customers.Any())
    return BadRequest();

  var mappedCustomer = Mapper.Map<IEnumerable<Customer>>(customers);
  customerRepository.AddCustomers(mappedCustomer);

  if (!customerRepository.SaveChanges())
    return StatusCode(StatusCodes.Status500InternalServerError, ″Impossibile
salvare le modifiche.″);

  return StatusCode(StatusCodes.Status201Created);
}
```

Il codice dell'Esempio 11.16 è molto simile a quello che è già stato introdotto in precedenza per l'inserimento di un singolo elemento. Le differenze si trovano nel fatto che la action risponde sulla route `/api/customerscollection` anziché `/api/customers` e nel fatto che con l'attributo `FromBody` si va a recuperare una intera collezione di elementi, non solo uno; come sempre sarà ASP. NET Core a fare il mapping secondo i formatter e secondo il `Content-Type` specificato a livello di richiesta. L'ultima differenza importante che possiamo notare nell'esempio riportato sopra è che, nel caso in cui sia andato a buon fine l'inserimento, anziché ritornare una `CreatedAtAction` che va a creare un header `Location` nella risposta e l'elenco degli oggetti creati, si ritorna solamente lo status code 201 (Created). Questa forse non è la pratica migliore, ma è questione di scelte: supponendo di dover caricare migliaia di dati, la risposta potrebbe diventare molto grande se dovessimo ricostruire tutti gli oggetti e ritornarli e, allo stesso modo,

l'URL generato nel `Location` header dovrebbe contenere tutti gli identificativi univoci delle risorse, tra l'altro facendo uso di un model binder personalizzato di cui non abbiamo ancora parlato ma che affronteremo più avanti nel corso del libro. L'elaborazione dei dati, però, non riguarda solamente la lettura o la creazione di nuovi elementi ma anche l'aggiornamento degli stessi, come vedremo a breve.

Eseguire aggiornamenti

Negli esempi illustrati in precedenza, si è sempre parlato di come affrontare un inserimento di una singola entità e di una collezione, oppure della lettura dei dati: ma come bisogna comportarsi in caso di un aggiornamento di una entità? Le strade da percorrere sono due e la scelta di una o dell'altra è dettata dalla logica di business. La prima strada possibile è quella che comporta l'aggiornamento completo della risorsa, come illustra l'Esempio 11.17, tramite l'uso del verb `PUT`.

Esempio 11.17

```
[HttpPut("{id}")]
public IActionResult UpdateCustomer(Guid id, [FromBody] CustomerForUpdate
customer)
{
  if (customer == null)
    return BadRequest();

  var mappedCustomer = Mapper.Map<Customer>(customer);
  mappedCustomer.Id = id;
  customerRepository.UpdateCustomer(mappedCustomer);

  if (!customerRepository.SaveChanges())
    return StatusCode(StatusCodes.Status500InternalServerError, "Impossibile
salvare le modifiche");

  return NoContent();
}
```

In questo esempio, la classe `CustomerForUpdate` è una variante di quelle viste in precedenza, per mantenere un certo livello di separazione: nel caso di un aggiornamento, infatti, per una scelta di logica, potremmo decidere che sia consentita la sola modifica dei campi nome e cognome. La proprietà `Id` che va a identificare in modo univoco il cliente da aggiornare viene recuperata dalla query string tramite il

matching dell'attributo `Id`, mentre il cliente arriva dopo la content negotiation dal body e, una volta rimappato con AutoMapper, viene eseguito l'aggiornamento, come mostrato nell'Esempio 11.18.

Esempio 11.18

```
public void UpdateCustomer(Customer customer)
{
  var internalCustomer = customers.FirstOrDefault(x => x.Id == customer.Id);
  internalCustomer.FirstName = customer.FirstName;
  internalCustomer.LastName = customer.LastName;
}
```

La richiesta inviata con Postman, in caso di cliente esistente si completerà correttamente, come ci si aspetta, con uno status code 204 (No Content), poiché non abbiamo specificato un tipo di ritorno per la action.

All'inivio con `PUT` di un oggetto di tipo `CustomerForUpdate` valido ma con la proprietà `LastName` non impostata, si otterrà una risposta con successo perché, a tutti gli effetti, non si sono verificati errori di alcun tipo e la validazione è stata superata senza problemi. Poiché viene rifatto l'assegnamento nell'`UpdateCustomer`, si otterrà un valore `null` anziché il valore precedente invariato.

Sebbene questo comportamento possa essere corretto in scenari come quello dell'esempio, in cui sono presenti solo due campi, in caso di logiche più complesse in cui si trattino dati con decine di proprietà, fare uso della `PUT` e reinviare ogni volta tutto il payload non è la soluzione migliore, anche a livello di banda. Per questo esiste la possibilità di fare aggiornamenti parziali tramite il verb `PATCH`.

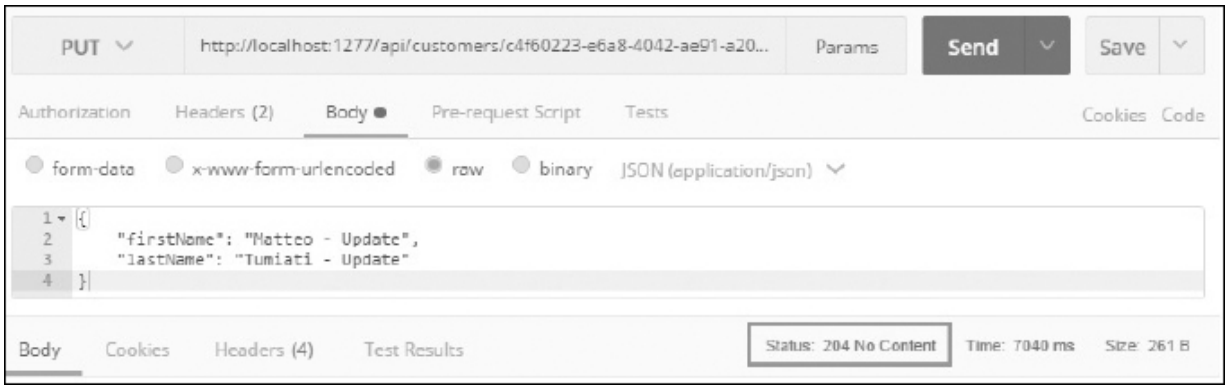


Figura 11.11 – L'aggiornamento completo dei dati di un cliente è possibile con una chiamata con verb `PUT` che restituirà come risultato 204 (No Content) in caso di update andato a buon fine.

L'operazione di `PATCH`, all'opposto della `PUT`, non accetta più in ingresso tutto l'oggetto `CustomerForUpdate` ma un oggetto di tipo `JsonPatchDocument`, che richiede anche un nuovo media-type `application/json-patch+json` per essere gestito. Si tratta di un file JSON, che include un elenco di operazioni che possiamo effettuare sugli oggetti inviati:

- `Add`: consente l'aggiunta di una nuova proprietà con il valore specificato, in casi in cui si stia lavorando con proprietà di tipo dynamic.
- `Remove`: invalida il contenuto della proprietà selezionata e lo imposta al suo valore di default (per esempio `null`).
- `Replace`: aggiorna il valore della proprietà selezionata con quella specificata nel documento.
- `Copy`: copia il valore di una proprietà verso un'altra.
- `Move`: è come l'operazione di `Copy` ma, in più, effettua anche una `Remove` sul percorso specificato.
- `Test`: verifica che il valore contenuto nel percorso richiesto sia identico a quello specificato nella richiesta.

Un JSON di un documento di questo tipo è disponibile nell'Esempio 11.19.

Esempio 11.19

```
[
    {
      ″op″: ″replace″,
      ″path″: ″/firstName″,
      ″value″: ″Matteo (Updated)″
    }
]
```

L'Esempio 11.19 mostra come deve essere costruito il documento che deve essere inviato. In particolare, possiamo notare come venga richiesta, tramite la proprietà `op`, la sostituzione del valore contenuto in `FirstName` con il valore contenuto nella proprietà `value`. Poiché è a tutti gli effetti un array, si possono concatenare diverse operazioni da effettuare in sequenza ma, per semplicità e per capirne il funzionamento, è più che sufficiente elaborare un solo parametro. L'Esempio 11.20 mostra un nuovo metodo capace di accettare un documento di questo tipo.

Esempio 11.20

```
[HttpPatch(″{id}″)]
public IActionResult PartialUpdateCustomer(Guid id, [FromBody] JsonPatchDocumen
t<CustomerForUpdate> patchDocument)
{
  if (patchDocument == null)
    return BadRequest();

  var customer = customerRepository.GetCustomer(id);
  var mappedCustomer = Mapper.Map<CustomerForUpdate>(customer);
  patchDocument.ApplyTo(mappedCustomer);

  customer = Mapper.Map<Customer>(mappedCustomer);
  customer.Id = id;
  customerRepository.UpdateCustomer(customer);

  if (!customerRepository.SaveChanges())
    return StatusCode(StatusCodes.Status500InternalServerError, ″Impossibile
salvare le modifiche″);

  return NoContent();
}
```

A livello di logica il flusso d'esecuzione di una chiamata PATCH è simile a quello già visto per una chiamata con PUT, ma ci sono un paio di differenze: la prima è che a livello di body non arriva più l'oggetto CustomerForUpdate ma un JsonPatchDocument<CustomerForUpdate>. La seconda differenza è che prima di richiamare l'aggiornamento sul repository viene applicata la modifica sul JsonPatchDocument, pertanto tutte le proprietà specificate nel documento verranno aggiornate, eliminate o aggiunte, mentre le altre rimarranno invariate, andando di fatto a risolvere il problema posto dalla PUT.

Con la gestione degli aggiornamenti va a concludersi l'overview relativa alle operazioni sui dati che sono necessarie per realizzare un sistema basato su WebAPI ma, come abbiamo già detto più volte, realizzare un'API non è la stessa cosa di scrivere un servizio RESTful. Pertanto ora vediamo come integrare le parti mancanti per avere un servizio completo.

Hypermedia As The Engine Of Application State (HATEOAS)

È già stato anticipato all'inizio del capitolo: lavorare e costruire delle WebAPI funzionali e funzionanti non è detto che coincida con lo sviluppo di un vero e proprio servizio RESTful. Il pezzo mancante alle WebAPI costruite per diventare un servizio REST a tutti gli effetti è la parte descrittiva perché, per quanto la parte server sia ben strutturata, è difficile pensare di avere un client che agisca ed evolva in modo completamente indipendente dalla parte server. Quali operazioni possiamo fare su una determinata risorsa? Possiamo aggiornare il dato in modo completo o anche in modo parziale? È possibile eliminare una risorsa? È possibile avere la paginazione nel caso in cui ci siano troppi dati da leggere? Queste sono solo alcune delle domande alle quali bisogna dare una risposta con un sistema auto-descrittivo perché, al momento, la logica presente nelle demo precedenti va solamente a lavorare con i dati e non con i metadata.

HATEOAS (Hypermedia As The Engine Of The Application State) è una serie di principi già ampiamente utilizzati in ambiente web e che

consente di aggiungere in risposta una serie di link, o di hypermedia, come se fossero dei metadati: in HTML, l'equivalente sarebbe l'anchor element, che include tramite `href` l'indirizzo sul quale recuperare la risorsa, in `rel` come si relaziona alla risorsa stessa e infine `type`, opzionale, che include il media-type. Tradotto in codice, quello che serve è una classe che gestisca queste tre nuove proprietà, come viene mostrato nell'Esempio 11.21.

Esempio 11.21

```
public class Link
{
  public string Href { get; set; }
  public string Rel { get; set; }
  public string Method { get; set; }
}
```

Nel caso delle WebAPI, il media-type viene gestito in un modo alternativo, come abbiamo già visto in precedenza in questo stesso capitolo, per questo è necessario interessarsi maggiormente su quale verb (o metodo) HTTP può essere invocato su un determinato URL. Poiché sono gli oggetti ritornati a dover includere i link, è necessario includere la proprietà corrispondente nel modello oppure estendere l'esistente. L'Esempio 11.22 mostra come fare.

Esempio 11.22

```
public class CustomerForReadExtended : CustomerForRead
{
  public ICollection<Link> Links { get; set; } = new List<Link>();
}
```

A questo punto, bisogna trovare un metodo per ricostruire il link corretto a partire da una action, ma per fortuna questo lavoro è già implementato dalla classe `UrlHelper` di ASP.NET Core, che va registrata appositamente come servizio aggiuntivo allo startup, come viene mostrato nell'Esempio 11.23.

Esempio 11.23

```
public void ConfigureServices(IServiceCollection services)
{
  // ...
  services.AddSingleton<IActionContextAccessor, ActionContextAccessor>();
  services.AddScoped<IUrlHelper>(factory =>
  {
    var actionContext = factory.GetService<IActionContextAccessor>()
                                        .ActionContext;
    return new UrlHelper(actionContext);
 });
}
```

L'interfaccia `IUrlHelper` della classe corrispondente può quindi essere iniettata nel costruttore del `CustomerController` e utilizzata per aggiungere i link corretti all'oggetto `Customer`, così da recuperare correttamente gli URL, come mostrato nell'Esempio 11.24.

Esempio 11.24

```
private CustomerForReadExtended AddLinks(CustomerForReadExtended customer)
{
  customer.Links.Add(new Link(urlHelper.Link("GetCustomer", new { id =
customer.Id }), "self", "GET"));
  customer.Links.Add(new Link(urlHelper.Link("CreateCustomer", null), "self",
"POST"));
  customer.Links.Add(new Link(urlHelper.Link("UpdateCustomer", new { id =
customer.Id }), "update_customer", "PUT"));
  customer.Links.Add(new Link(urlHelper.Link("PartialUpdateCustomer", new { id =
customer.Id }), "partial_update_customer", "PATCH"));

  return customer;
}
```

Come possiamo notare nell'Esempio 11.24, vengono aggiunti i link nella lista prevista dall'oggetto `Customer` e vengono costruiti partendo dalle proprietà `Name` aggiunte sopra le corrispettive action, come viene illustrato nell'Esempio 11.25.

Esempio 11.25

```
[HttpGet("{id}", Name = "GetCustomer")] public IActionResult GetCustomer(Guid id)
{
}
```

Per costruire correttamente gli URL, è necessario passare anche eventuali parametri, come per esempio gli identificativi dei clienti, quindi andiamo ad aggiungere i valori relativi a `Rel`, e `Method` (o `Type`): `Rel` è rappresentato dal valore `SELF` nei casi in cui la chiamata venga fatta sulla stessa URL del chiamante, mentre ha un altro valore altrettanto descrittivo quando si riferisce ad action differenti. La proprietà `Method` è valorizzata con il corrispettivo verb HTTP che la chiamata si aspetta (per esempio, `POST` in caso ci si riferisca all'URL di creazione di un nuovo oggetto). L'ultimo passaggio da eseguire è quello di cambiare la risposta per la action `GetCustomer`, come viene evidenziato nell'Esempio 11.26.

Esempio 11.26

```
[HttpGet("{id}", Name = "GetCustomer")]
public IActionResult GetCustomer(Guid id)
{
  var customer = customerRepository.GetCustomer(id);
  if (customer == null)
    return NotFound();
  var customerForRead = Mapper.Map<CustomerForReadExtended>(customer);
  return StatusCode(StatusCodes.Status200OK, AddLinks(customerForRead));
}
```

Il risultato ottenuto è visibile infine nella Figura 11.12, che illustra perfettamente come siano visibili tutti i link necessari per realizzare altre richieste al server sulla stessa risorsa di tipo cliente.

GET http://localhost:1277/api/customers/46c46329-dd68-4e01-8e57-a1c... Params Send Save

Authorization Headers (2) Body Pre-request Script Tests Cookies Code

Body Cookies Headers (6) Test Results Status: 200 OK Time: 2355 ms Size: 856 B

Pretty Raw Preview JSON


```
1  {
2      "links": [
3          {
4              "href": "http://localhost:1277/api/customers/46c46329-dd68-4e01-8e57-a1c4d026f98d",
5              "rel": "self",
6              "method": "GET"
7          },
8          {
9              "href": "http://localhost:1277/api/customers",
10             "rel": "self",
11             "method": "POST"
12         },
13         {
14             "href": "http://localhost:1277/api/customers/46c46329-dd68-4e01-8e57-a1c4d026f98d",
15             "rel": "update_customer",
16             "method": "PUT"
17         },
18         {
19             "href": "http://localhost:1277/api/customers/46c46329-dd68-4e01-8e57-a1c4d026f98d",
20             "rel": "partial_update_customer",
21             "method": "PATCH"
22         }
23     ],
24     "id": "46c46329-dd68-4e01-8e57-a1c4d026f98d",
25     "name": "Matteo Tumiati",
26     "age": 27
27 }
```


Figura 11.12 – I link in risposta a una chiamata in `GET` fungono da metadati per il self-discovery lato client e vengono inviati come parte del payload di risposta.

Nonostante questa tecnica sia piuttosto diffusa nel mondo delle REST API, implica la creazione di una funzione personalizzata per ogni metodo. In alternativa o, meglio, in aggiunta a quanto già realizzato con i link, ci sono ancora due verb HTTP che possiamo sfruttare per dare una maggior indicazione del servizio, in modo tale che si possa fare auto-discovery degli endpoint: si tratta di `OPTIONS` e `HEAD`.

Una richiesta inviata come `OPTIONS` è utile perché il client può fare il discovery dei metodi che sono abilitati a rispondere su un determinato URL, senza però ricevere in risposta l'effettivo risultato. In ASP.NET Core WebAPI viene gestita come è evidenziato nell'Esempio 11.27.

Esempio 11.27

```
[HttpOptions]
public IActionResult GetCustomersOptions()
{
  Response.Headers.Add("Allow", "GET,POST,OPTIONS");
  return Ok();
}
```

La chiamata all'endpoint `/api/customers` con il verb `OPTIONS` non fa altro che aggiungere un nuovo header, chiamato `Allow`, in cui sono elencati in forma di stringa e separati da virgola tutti i verb HTTP che rispondono sullo stesso endpoint. In questo caso specifico, come possiamo notare, non vengono elencati `DELETE`, `PATCH` e `PUT` poiché questi in realtà rispondono su un URL differente, che include anche l'identificativo del cliente, ed è proprio per questo motivo che questo metodo andrebbe usato assieme a quello dei link visti in precedenza. In qualsiasi caso, anche in quelli in cui non ci sono verb HTTP che supportano l'endpoint richiesto, la chiamata deve rispondere con uno status code di 200 (OK) se la action è marcata con l'attributo `HttpOptions`. Il risultato di una chiamata è evidenziato nella Figura 11.13.

Figura 11.13 – L'header `Allow` mostra l'elenco dei metodi HTTP che sono utilizzabili sull'URL appena chiamato.

La chiamata con verb `HEAD` invece, può essere fatta su tutte le action che dichiarano anche altri verb, come `GET` per esempio, e viene utilizzata

principalmente per testare se i parametri passati in ingresso (query string, header e body) e le header in uscita sono validi per poter considerare l'integrazione dell'API lato client come funzionante: non verrà ritornato, infatti, alcun contenuto in risposta, anche per le chiamate in `GET`, proprio perché non è necessario e sarà direttamente il runtime a gestire l'output, senza che ci sia la necessità di farlo all'interno della action. L'unica modifica è quella di marcare la funzione corrispondente con l'apposito attributo, come viene mostrato nell'Esempio 11.28.

Esempio 11.28

```
[HttpGet]
[HttpHead]
public IActionResult GetCustomers()
{
  // ...
}
```

La stessa API che risponde all'URL `/api/customers` supporterà chiamate sia in `GET` sia in `HEAD`, come viene evidenziato nella Figura 11.14, ma non produrrà alcun output.

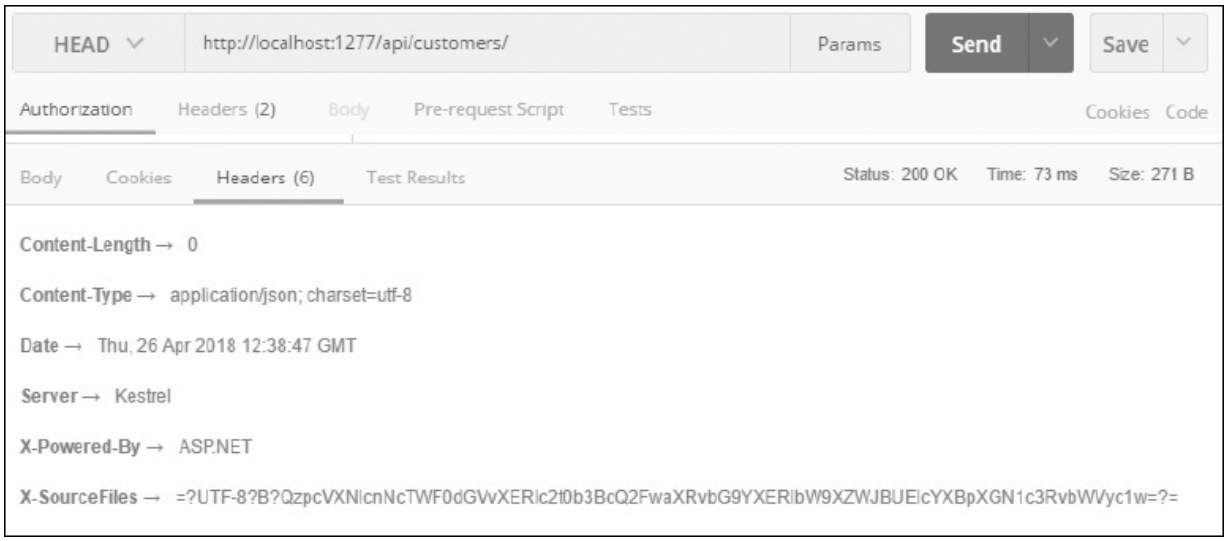


Figura 11.14 – Una chiamata con verb `HEAD` non restituisce alcun risultato ma mantiene tutti gli header e gli status code in risposta.

Nonostante non venga riportato il body nella risposta, sarà comunque possibile, grazie al servizio di self-discovery e in aggiunta alla chiamata fatta in precedenza con `OPTIONS`, avere un sistema di WebAPI completamente autonomo, testabile e riproducibile.

Conclusioni

In questo capitolo sono stati trattati due temi di eguale importanza che riguardano il tema della comunicazione: il primo, forse il più utilizzato secondo le esigenze odierne, è il mondo delle WebAPI, che permette di esporre come servizio su un preciso endpoint i dati che vengono prelevati direttamente dal database o da una qualsiasi altra fonte di dati attraverso uno o più strati applicativi in modo proporzionale alla complessità del proprio sistema, anche se per semplicità abbiamo solamente affrontato il tema del repository pattern. Successivamente, abbiamo parlato delle differenze che ci sono nel lavorare con le WebAPI, standard di ASP.NET Core, e quali sono i vantaggi nel trasformare queste ultime, facendo anche uso di HATEOAS in RESTful API, per avere maggiore separazione tra client e server, pur mantenendo quelli che sono i principi base di funzionamento. Una volta preparato il design e l'architettura delle API, è necessario pensare a come effettuarne il mantenimento e il versionamento.

Nel prossimo capitolo vedremo degli scenari avanzati e discuteremo non solo quali strategie e quali strumenti possano essere utilizzati per quest'ultimi due argomenti, ma vedremo anche i dettagli di un nuovo sistema di comunicazione, i WebHook, e parleremo delle novità riguardanti una infrastruttura per comunicazione in tempo reale.

12

Sviluppare servizi RESTful avanzati con ASP.NET Core WebAPI, WebHook e SignalR

Nel corso del capitolo precedente abbiamo introdotto quelli che sono i concetti relativi alle WebAPI e, in particolare, abbiamo parlato di come ASP.NET si sia evoluto per integrare nel framework di MVC anche tutta la parte dei `WebApiController`. Abbiamo inoltre discusso di quali sono le differenze tra lo sviluppo di una WebAPI standard, comunque accettabile e funzionale, e creare un vero e proprio servizio RESTful, introducendo concetti relativi all'uso dei metadati per supportare HATEOAS e avere il massimo dell'indipendenza, per permettere sviluppi concorrenti tra client e server.

Sempre in quest'ottica nascono i WebHook, ovvero delle callback HTTP che permettono l'estensione di una WebAPI con un'altra, invocata in modo asincrono e che potrebbe appartenere a vendor differenti, consentendo, di fatto, l'estensione del servizio anche lato server. Ci sono invece casi, all'opposto di questi, in cui il client ha la necessità di comunicare senza latenza con il server, come per esempio servizi esposti dal mondo della finanza oppure, tipicamente, i giochi. Nonostante lo scenario sembri avanzato e complesso, nel corso del capitolo parleremo di come uno strumento come SignalR, integrato in ASP.NET Core, venga incontro alle esigenze di tutti gli sviluppatori per semplificare gli scenari di comunicazione real-time.

Una volta costruito tutto il sistema di comunicazione tramite API, WebHook e SignalR, così come per tutto quello che riguarda il mondo del software, bisogna pensare alla manutenzione: è fondamentale prevedere il servizio come aggiornabile per quei casi in cui ci sia un cambio del

modello dei dati, oppure nei meccanismi di elaborazione, pertanto è necessario lavorare con un sistema che supporti il versionamento. Infine, imparare a scrivere la corretta documentazione ed esporla a sua volta come se fosse un servizio, in costante aggiornamento, è d'obbligo per farsi immediatamente un'idea del contesto e funzionamento delle API stesse.

Documentazione con Swagger

La scrittura delle API è completata ma, come abbiamo anticipato e da buone abitudini da sviluppatori, è bene iniziare a pensare alla documentazione per lasciare traccia di quello che è stato fatto e del funzionamento per i vari client che andranno a utilizzare questi servizi. Scrivere la documentazione è un processo tedioso che difficilmente rende felici gli sviluppatori ma, per fortuna, ci sono diversi strumenti che vengono in aiuto per cercare di automatizzare il più possibile l'intero processo. Tra questi strumenti ci sono Swagger, una specifica utilizzata per documentare le API che può essere scritta in formato JSON o YAML, e Swashbuckle, uno strumento che permette di creare la documentazione per Swagger in modo automatico a partire dai commenti XML inseriti nel codice. Maggiori informazioni e la relativa documentazione su Swagger sono disponibili su `http://aspit.co/bno`.

Questi due strumenti sono già integrati in un pacchetto di NuGet da aggiungere alla soluzione, chiamato `Swashbuckle.AspNetCore`. Il passaggio successivo, come spesso accade, è la registrazione del servizio all'interno della `ConfigureServices`, come è visibile nell'Esempio 12.1.

Esempio 12.1

```
services.AddSwaggerGen(c =>
{
  c.SwaggerDoc("v1", new Info
  {
    Version = "v1",
    Title = "Documentazione WebAPI ASP.NET Core",
    Description = "Esempio di come si fa la documentazione con ASP.NET Core",
    TermsOfService = "None",
    Contact = new Contact
    {
      Name = "ASPItalia.com",
```

```
        Email = string.Empty,
        Url = ″https://aspitalia.com″
      }
    });
    var xmlFile = $″{Assembly.GetEntryAssembly().GetName().Name}.xml″;
    var xmlPath = Path.Combine(AppContext.BaseDirectory, xmlFile);
    c.IncludeXmlComments(xmlPath);
  });
```

I parametri definiti all'interno di questa funzione possono essere personalizzati secondo le nostre esigenze. Escludendo la configurazione di base, possiamo notare come venga anche passato un file XML recuperato dalla directory di output (o di publish): questo file contiene tutti i commenti XML inseriti sui vari metodi nel codice e deve essere abilitato in modo esplicito nella pagina Build delle proprietà del progetto, come è illustrato nella Figura 12.1.

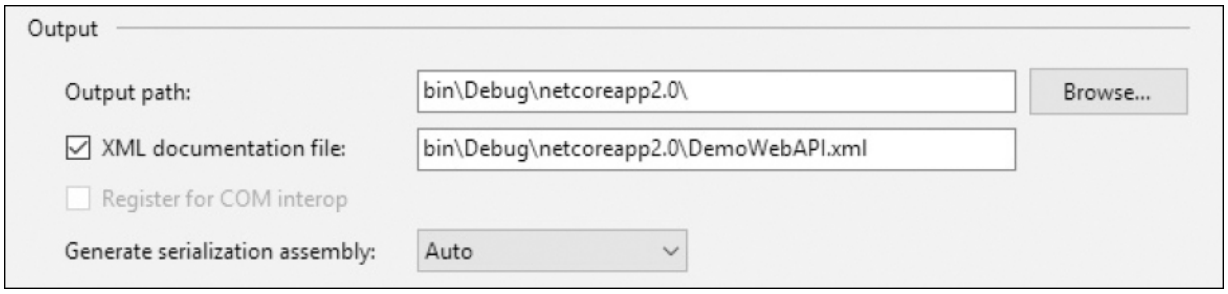


Figura 12.1 – Dalle proprietà del progetto, all'interno di Visual Studio, è possibile abilitare la pubblicazione della documentazione in formato XML nella cartella di output della build o in un qualsiasi altro percorso.

La parte più interessante arriva quando si vuole avviare il servizio: nei paragrafi precedenti è stato anticipato come Swagger funzioni su una specifica JSON o YAML, che non sono facilmente leggibili nel caso in cui ci siano molte API, ma per fortuna Swashbuckle contiene, fra le altre funzionalità, anche un motore che consente di generare dell'interfaccia grafica a partire dal file di Swagger, in modo molto intelligente e intuitivo, garantendo anche interattività con le varie API. Per abilitare l'interfaccia è sufficiente registrare Swagger e SwaggerUI nel metodo `Configure`. L'Esempio 12.2 mostra una tipica registrazione in tal senso.

Esempio 12.2

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
  // ...
  app.UseStaticFiles();
  app.UseSwagger();
  app.UseSwaggerUI(c =>
  {
    c.SwaggerEndpoint("/swagger/v1/swagger.json", "Versione 1.0");
  });
  app.UseMvc ();
}
```

Come possiamo notare, nell'Esempio 12.2, è anche necessario abilitare l'uso dei file statici, altrimenti non sarà possibile leggere il file `swagger.json`. Avviando l'applicazione e navigando alla route `/swagger`, è già possibile vedere l'interfaccia grafica, come è mostrato nella Figura 12.2.

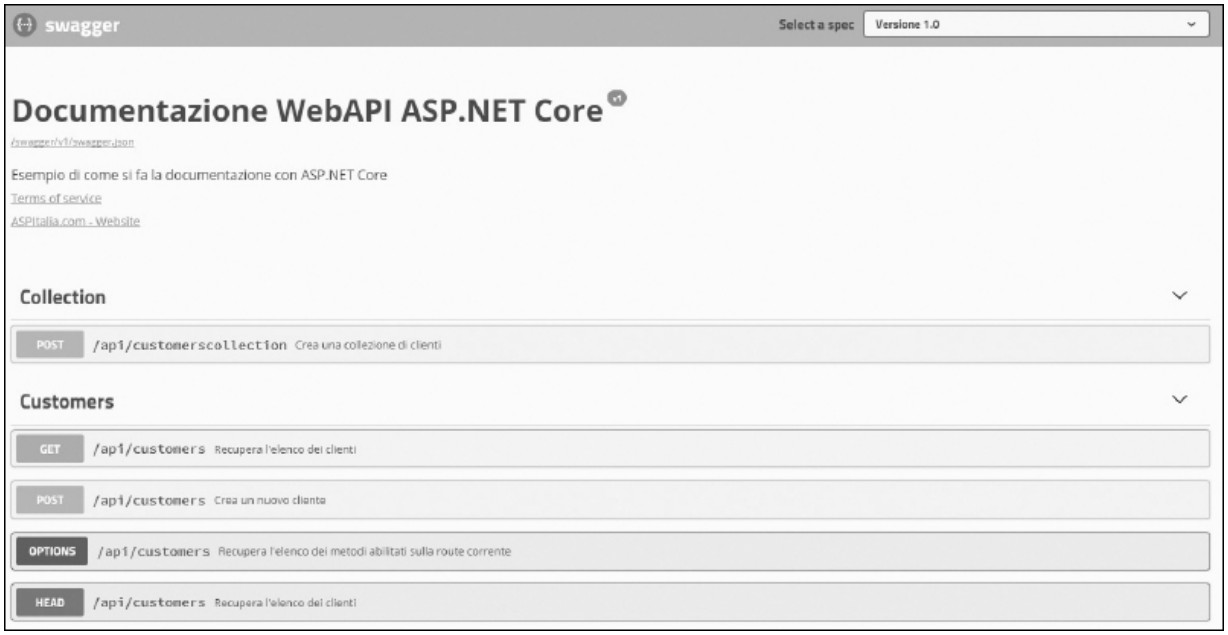


Figura 12.2 – Navigando all'URL localhost/swagger è possibile vedere l'interfaccia grafica esposta da Swagger, costruita sopra il file JSON posizionato in localhost/swagger/v1/swagger.json, oppure esposto nel percorso specificato in `SwaggerEndpoint`.

Ogni singolo elemento esposto in questa interfaccia è stato recuperato dal file XML generato in automatico dai commenti: pertanto, non solo saranno visibili le route ma sarà anche possibile vedere i tipi di ritorno, gli errori generati e disporre di una dashboard per provare le API.

La action in cui recuperiamo un cliente esistente è stata pertanto modificata dal capitolo precedente con i commenti illustrati nell'Esempio 12.3.

Esempio 12.3

```
/// <summary>
/// Recupera un cliente
/// </summary>
/// <param name="id">L'identificativo di un cliente</param>
/// <returns>Un cliente selezionato tramite il suo identificativo</returns>
/// <response code="200">Ritorna il cliente scelto tramite il suo id</response>
/// <response code="404">Se il cliente non è stato trovato</response>
[HttpGet("{id}", Name = "GetCustomer")]
[Produces("application/json", "application/xml")]
[ProducesResponseType(typeof(CustomerForRead), 200)]
[ProducesResponseType(404)]
public IActionResult GetCustomer(Guid id)
{
  var customer = customerRepository.GetCustomer(id);

  if (customer == null)
    return NotFound();

  var customerForRead = Mapper.Map<CustomerForRead>(customer);

  return StatusCode(StatusCodes.Status200OK, customerForRead);
}
```

Oltre ai commenti più classici come il `summary`, il `return` ed eventuali parametri, sono comparsi i commenti di `response code`: vengono utilizzati da Swagger per mostrare i potenziali tipi di risposta. La Figura 12.3 mostra un esempio di risposta prodotta a partire dal codice precedente.

Code	Description

200	*Ritorna il cliente selezionato tramite il suo identificativo*
	
	Example Value | Model
	
	{
	"id": "string",
	"name": "string",
	"age": 0
	}

404	*Se il cliente non è stato trovato*

Figura 12.3 – La combinazione dei commenti XML nel tag `response` **e degli attributi** `ProducesResponseType` **viene letta da Swagger per mostrare in maniera visiva e naturale le possibili risposte dell'API.**

A livello di attributi, invece, possiamo notare `Produces`, che indica quale media-type ci possiamo aspettare in ingresso e in risposta per la content negotiation, e `ProducesResponseType`, che specifica la tipologia di modello, ovvero la classe che ci si deve aspettare nel caso in cui ci sia una risposta con un determinato status code, solitamente 200 (OK).

Figura 12.4 – Le classi C# identificate nei parametri delle action vengono esposte da Swagger per essere utilizzate direttamente tramite l'interfaccia grafica e fare un controllo d'integrità dei tipi prima dell'invio della richiesta al server.

Nel caso in cui si voglia ottenere un'interfaccia grafica più coerente con il resto del sito web costruito, è anche possibile andare a modificare i CSS e crearne di personalizzati: è necessario creare la cartella swagger/ui all'interno di wwwroot e aggiungere i nuovi fogli di stile. Una volta organizzata ed eventualmente personalizzata la documentazione iniziale, non sarà complicato averla sempre aggiornata nel tempo, poiché richiede, tipicamente, la sola scrittura di poche righe di commento per ogni action che viene costruita. L'aggiornamento stesso della documentazione non è nemmeno detto che sia così frequente ma dipende principalmente da quanto le API stesse vengono modificate nel tempo. Per questo è utile iniziare a parlare di come vengono mantenute le versioni.

Gestione delle versioni

Nel corso degli anni un'API può evolvere, così come il codice dell'intera applicazione. I servizi REST, però, di fatto stabiliscono un contratto (anche se poco formale) e non andrebbe mai rimosso un endpoint, o, peggio ancora, variato il comportamento.

Possiamo segnalare questi casi attraverso il versioning, applicando una sorta di tag alle API per specificare che l'implementazione rimarrà identica, fino al momento in cui verrà resa deprecata ed eventualmente dismessa, ma le modifiche al modello o agli strati applicativi (per esempio quelli per recuperare i dati dal database) possono ancora esserci: per questi verrà applicato un tag diverso o, per essere più precisi, una versione maggiorata, che marcherà i servizi su nuovi endpoint.

ASP.NET Core di default non ha qualcosa di già implementato ma ci sono ben tre applicazioni che possiamo integrare:

- Versioning tramite parametri della query string: sfruttando un attributo personalizzato, per esempio `api-version`, possiamo effettuare una richiesta a un determinato URL con una specifica versione.

- Versioning sull'header: inviando un attributo personalizzato, per esempio `X-API-Version`, che può essere utilizzato per tenere traccia della versione corrente dell'API.

- Versioning sull'URL: è il link stesso a contenere una route o un host differente, per indicare il numero di versione utilizzato.

Poiché per realizzare la seconda soluzione è necessario avere un'approfondita conoscenza dei middleware, che affronteremo nei capitoli successivi, e considerando che la terza soluzione è troppo complessa, per cui la vedremo solo in parte, per il momento adotteremo la prima soluzione che, seppure non è la più elegante, è comunque funzionale.

La prima cosa da fare è aggiungere da NuGet il pacchetto `Microsoft.AspNetCore.Mvc.Versioning` che va a integrare tutte le

classi necessarie per gestire il versionamento, per poi registrare il servizio al momento dello startup nella `ConfigureServices`, come mostrato nell'Esempio 12.4.

Esempio 12.4

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddApiVersioning(config =>
  {
    config.AssumeDefaultVersionWhenUnspecified = true;
    config.DefaultApiVersion = new ApiVersion(1, 0);
  });
}
```

Durante la fase di configurazione è stato necessario specificare che la versione di default utilizzata è la "1.0", questo anche per via del fatto che è l'unica attualmente supportata. Inoltre, è stata anche impostata su `true` la proprietà `AssumeDefaultVersionWhenUnspecified` che permette di fare il fallback in automatico alla versione di default qualora non dovesse essere specificata nella query string. Per testarne il funzionamento, è importante impostare i due controller sulla stessa route (per fare in modo che si chiamino nello stesso nome è sufficiente cambiare il nome del namespace), come viene mostrato nell'Esempio 12.5.

Esempio 12.5

```
namespace DemoWebAPI.Controllers
{
    [ApiVersion(″1.0″)]
    [Route(″api/versioning″)]
    public class VersioningController : Controller
    {
      [HttpGet]
      public IActionResult Get()
      {
        return Ok(″v1″);
      }
    }
}
namespace DemoWebAPI.Controllers.V2
{
    [ApiVersion(″2.0″)]
```

```
    [Route("api/versioning")]
    public class VersioningController : Controller
    {
      [HttpGet]
      public IActionResult Get()
      {
        return Ok("v2");
      }
    }
}
```

Nell'Esempio 12.5 possiamo notare come entrambe le API abbiano sopra l'attributo `Route` un nuovo attributo, che introduciamo ora, chiamato `ApiVersion`. Al contrario di quello che avviene in fase di startup, l'attributo `ApiVersion` accetta una stringa, quindi è fondamentale prestare attenzione durante la scrittura delle versioni, altrimenti si potrebbe correre il rischio di fare sempre fallback sulla versione di default. La chiamata fatta specificando (o non) in query string il valore `api-version` farà variare la risposta da "v1" a "v2", come viene mostrato nell'immagine 12.5.

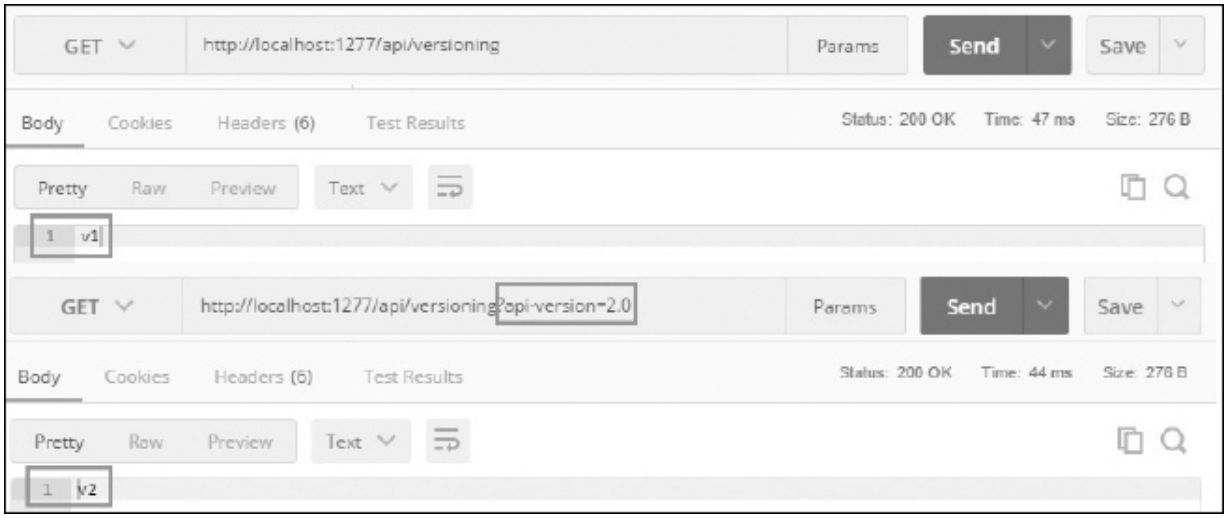


Figura 12.5 – La risposta alle API esposte sulla stessa route dipende dal parametro `api-version` specificato nella querystring.

In alternativa, come abbiamo anticipato, è possibile specificare la versione direttamente all'interno della route, anche se in questo caso particolare viene poi persa la possibilità di avere il fallback sulla versione

di default. Poiché viene gestito a un livello differente, in caso in cui non venga più specificata la versione, il risultato dell'API ritornerà un errore 404 (Not Found). L'Esempio di codice 12.6 dimostra come sia fattibile semplicemente aggiornando il percorso della route con l'apposito attributo.

Esempio 12.6

```
[Route(˝api/{version:apiVersion}/versioning˝)]
```

Quando il codice inizia a evolvere in più versioni, talvolta può aver senso cominciare a deprecare endpoint che non sono più validi, anche se questi vengono ancora supportati e manutenuti. Per questo esiste la possibilità di aggiungere la proprietà `Deprecated`, che manderà in risposta un header `api-deprecated-version` contenente il numero della versione deprecata, come viene mostrato nell'Esempio 12.7.

Esempio 12.7

```
[ApiVersion(˝1.0˝, Deprecated = true)]
```

Al contrario, può succedere che una determinata API sia mantenuta invariata nel tempo anche con il passare delle versioni su altri endpoint: per questo, al posto di specificare ogni volta il supporto a una nuova versione, può far comodo specificare l'attributo `ApiVersionNeutral`.

Arrivati a questo punto, le API possono già essere utilizzate, distribuite e mantenute nel corso del tempo. Per via di come sono costruite le API, ovvero in una sorta di micro-servizi, può succedere talvolta che ci sia la necessità di doverle estenderle per realizzare un vero e proprio sistema complesso e personalizzato secondo le nostre esigenze: per questo introduciamo il concetto di WebHook.

Gestire endpoint basati su WebHook

Abbiamo già parlato durante questo stesso capitolo e nel capitolo precedente di cosa sono e di quanto le WebAPI possano essere utili per consumare dei servizi da parte di un client. Talvolta, però, può essere necessario estendere la funzionalità stessa esposta dal servizio sfruttando altri servizi di terze parti, ed è qui che vengono in gioco i WebHook: si tratta di chiamate effettuate con metodo `POST`, che vengono richiamate al verificarsi di un determinato evento, detto trigger.

I WebHook funzionano come una callback HTTP e sono supportati nativamente da ASP.NET Core a partire dalla versione 2.1, con il supporto per diversi provider come Azure Alerts, Kudu, Dynamics CRM, Bitbucket, Dropbox, GitHub, MailChimp, Pusher, Salesforce, Slack, Stripe, Trello e WordPress, con integrazioni per altri servizi previste in un futuro prossimo.

Negli esempi che seguiranno, vedremo l'integrazione tramite GitHub, dato che è probabilmente il servizio più conosciuto di quelli esposti in precedenza, ma il funzionamento è piuttosto simile nel caso degli altri provider. Per iniziare, è necessario installare il pacchetto di NuGet `Microsoft.AspNetCore.WebHooks.Receivers.GitHub` e registrare l'apposito servizio nella `ConfigureServices` della classe `Startup`, come è mostrato nell'Esempio 12.8.

Esempio 12.8

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddMvc()
           .AddGitHubWebHooks();
}
```

Prima di ricevere le notifiche, l'altra cosa che bisogna fare all'interno del progetto ASP.NET Core è registrare una action che sia in grado di gestire la callback, come nel codice dell'Esempio 12.9.

Esempio 12.9

```
[GitHubWebHook]
public IActionResult Receiver(string id, string @event, JObject data)
```

```
{
  // ...
  return Ok();
}
```

Marcando la action `Receiver` con l'attributo `GitHubWebHook` facciamo in modo che sia in grado di gestire la route adeguata. I parametri esplicitati nella firma del metodo verranno riempiti con i valori che GitHub riterrà più opportuno e sono gli unici valori che possono variare tra WebHook esposti da provider differenti. GitHub potrà mandare una richiesta con un determinato payload riguardo a specifici eventi che avvengono su un repository specificato, come l'aggiunta di un nuovo commit, la creazione di un nuovo branch o di un nuovo tag, l'update di una issue o una nuova pull request, mentre per quanto riguarda Azure, per esempio, possiamo essere informati riguardo al superamento di una certa soglia di consumo: la funzione `Receiver` dovrà implementare la logica personalizzata secondo le esigenze del progetto e a seconda del trigger selezionato. All'interno delle impostazioni del repository di GitHub è possibile andare ad aggiungere un nuovo WebHook, come è illustrato nella Figura 12.6.

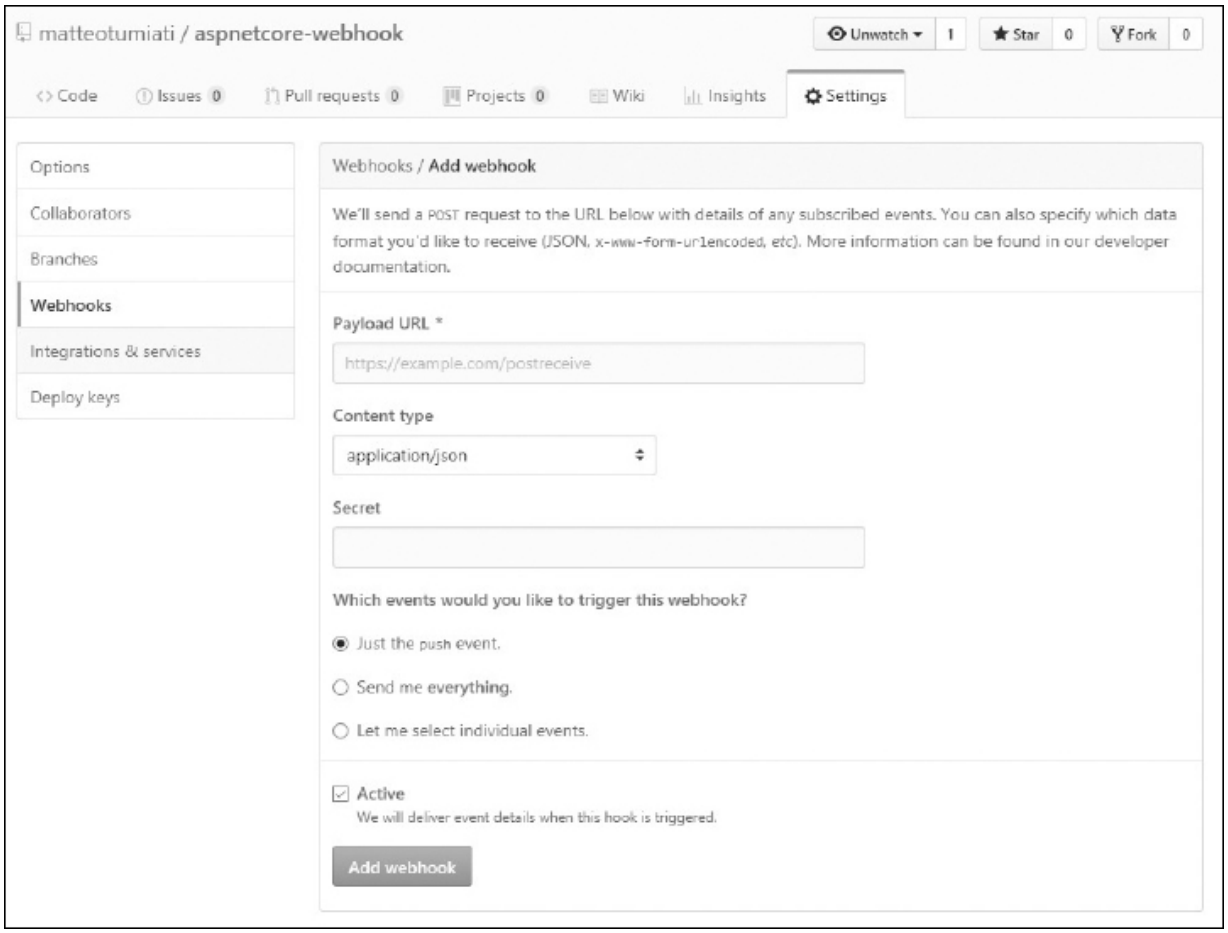


Figura 12.6 – Dal proprio repository di GitHub è possibile accedere alla configurazione dei WebHook tramite l'apposito menù all'interno delle impostazioni.

Poiché GitHub richiede un URL raggiungibile pubblicamente (e il localhost della macchina di sviluppo non lo è), dobbiamo affidarci a strumenti come ngrok, che permettono di esporre localhost su FQDN accessibili via internet. Per avviare ngrok è necessario lanciare dalla console di PowerShell il comando riportato nell'Esempio 12.10.

Esempio 12.10

```
.\ngrok.exe http localhost:{porta-esposta-da-aspnet-core}
```

Se l'ambiente richiesto è in grado di partire correttamente, il risultato ottenuto sarà simile a quello illustrato nella Figura 12.7.

```
Windows PowerShell
ngrok by @inconshreveable                                        (Ctrl+C to quit)

Session Status                online
Session Expires               7 hours, 59 minutes
Version                       2.2.8
Region                        United States (us)
Web Interface                 http://127.0.0.1:4040
Forwarding                    http://e83c0a4f.ngrok.io -> localhost:1277
Forwarding                    https://e83c0a4f.ngrok.io -> localhost:1277

Connections                   ttl     opn     rt1     rt5     p50     p90
                              0       0       0.00    0.00    0.00    0.00
```

Figura 12.7 – Dopo aver lanciato il comando di avvio, tramite ngrok sarà possibile vedere gli URL generati con forwarding su HTTP e HTTPS, oltre che il periodo di uptime e il numero di connessioni attive.

A questo punto, possiamo modificare la configurazione di GitHub andando a impostare i parametri nel modo seguente:

- Payload URL: è l'host name assegnato da ngrok unito al path `/api/webhooks/incoming/github`.
- Content-Type: per via dei formatter impostati nel capitolo precedente è necessario impostare `application/json`.
- Secret: un token generato con un qualsiasi tool per HMAC.

Lo stesso secret deve essere poi importato anche all'interno del progetto nel file di `appsettings.json`, come è visibile nell'Esempio 12.11.

Esempio 12.11

```
  "WebHooks": {
    "GitHub": {
      "SecretKey": {
        "default": "752a7cf3cd87c32b65a585b82fcf06ff9972f125"
```

```
      }
    }
  }
```

Questa configurazione verrà poi letta in fase di startup dell'applicazione ASP.NET Core e sarà effettuata la registrazione del WebHook dedicato. Creando una issue all'interno del repository oppure facendo una nuova commit sul file, si potrà vedere, debuggando il progetto, il flusso di dati entrare all'interno della funzione definita in precedenza.

Al contrario di quanto viene introdotto dal concetto dei WebHook, ovvero che è bene fare separazione tra più servizi ed estenderli in un secondo momento, è bene notare come ci siano scenari in cui avere il dato nel minor tempo possibile è fondamentale per produrre immediatamente una notifica. Per questo, client e server devono essere fortemente accoppiati in modo da ridurre al massimo la latenza e c'è il bisogno di avere un framework, come SignalR, che semplifichi lo sviluppo.

Comunicazione in tempo reale con SignalR

All'interno di questo capitolo sono stati affrontati diversi meccanismi di comunicazione, partendo dalle WebAPI, asincrone e invocate su richiesta, ai WebHook, utilizzati principalmente per estendere il funzionamento di un determinato servizio attraverso una o più WebAPI. Nel capitolo precedente, invece, sono anche stati anche discussi quelli che sono i diversi vantaggi di avere una forte separazione tra il client e il server che espone il servizio, parlando per esempio di soluzioni basate su self-discovery ma, così come ci sono scenari in cui questo ha senso, ci sono altrettanti scenari in cui è necessario avere l'esatto opposto: ricerche o aggiornamenti di dati, le azioni e le operazioni finanziarie in genere, notifiche di un gol durante una partita, giochi online o applicazioni che richiedono della collaborazione in tempo reale (per esempio Word Online) sono solo alcuni degli scenari che richiedono una latenza minima nello scambio dei messaggi. Sebbene si possa pensare che queste siano applicazioni molto specifiche, in realtà non lo sono; infatti, il concetto di notifica può essere applicato a qualsiasi ambiente,

anche per applicazioni di campo industriale, per notificare l'avvenuta produzione di un determinato pezzo oppure l'aggiornamento sulla quantità dei prodotti venduti e così via. Inoltre, poiché principalmente il tutto viene riportato ad uso e consumo degli utenti, sorgono altri problemi: gli utenti vogliono vedere sempre tutto aggiornato, nel più breve tempo possibile e su ogni device con ogni tipo di connessione.

Il paradigma cambia quindi in modo drastico e di fatto non si parlerà più di client e server ma di publisher e subscriber: il subscriber, o consumer, è chi vuole iscriversi a un servizio esposto dal publisher per essere notificato all'avvenimento di un evento, mentre il publisher è colui che tiene traccia di tutti i consumer, che può identificare in modo univoco, in modo da interrogarli e di poter mandare loro i dati allo scatenarsi di un trigger, quale, per esempio, la modifica di un dato in un database, come si può vedere nell'architettura illustrata nella Figura 12.8.

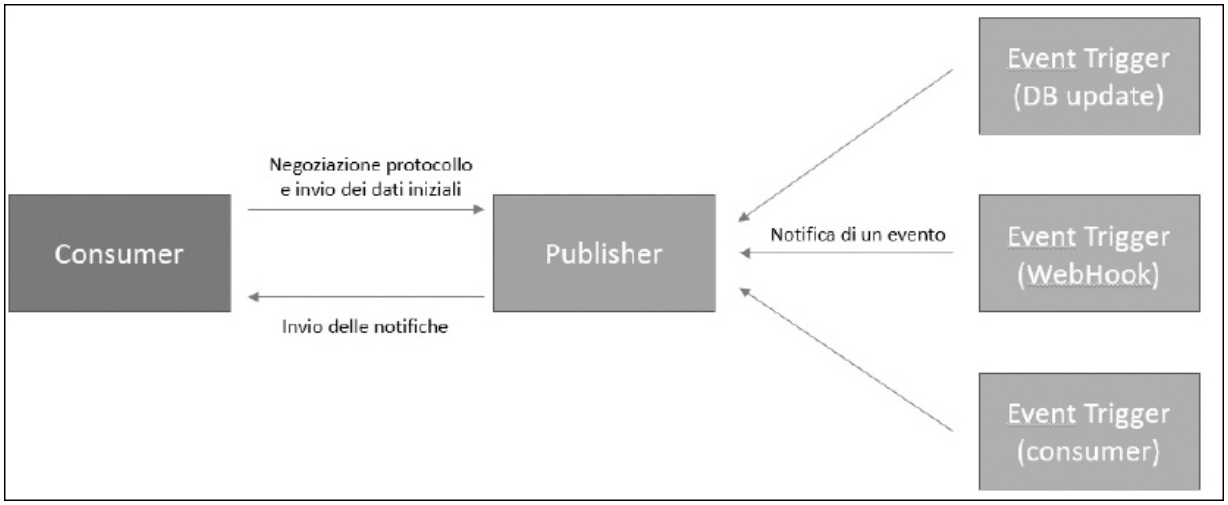


Figura 12.8 – L'architettura basata sul "pattern" di publisher e subscriber elimina di fatto la differenza tra client e server.

Le notifiche che vengono inviate dal publisher sono volatili, per cui non è previsto alcun sistema di persistenza o di invii multipli della stessa notifica qualora non ci sia un ack da parte del subscriber per notificare al subscriber la ricezione della notifica: la priorità è che la notifica deve arrivare nel più breve tempo possibile, altrimenti non avrebbe senso

mandarla, per cui non c'è spazio per fare altre operazioni oltre all'invio del singolo messaggio. Nel caso in cui ci sia, invece, la necessità di persistere i messaggi, assicurarsi che arrivino a tutti i subscriber e che vengano gestiti tramite appositi ambienti "poisoned" quando non vengono prelevati dal consumer dopo un certo periodo temporale, allora è necessario puntare su meccanismi di code come Service Bus piuttosto che Azure Queue.

A livello di comunicazione, invece, sorge un nuovo problema poiché tutto, come abbiamo già visto, è attualmente basato su HTTP, un protocollo di request-response, non publisher-subscribe: come si può rendere compatibili questi due sistemi? Per dare una risposta a questa domanda sono nati diversi meccanismi e protocolli che funzionano al di sopra di HTTP:

- `Periodic polling`: il client chiede a intervalli regolari al server se ci sono novità riguardo a un determinato evento, e il server continua a rispondere con un payload indicante il risultato della richiesta effettuata dal client.

- `Long polling`: con l'introduzione di Comet, un modello architetturale basato sugli eventi server-side, non c'è più bisogno che il client richieda in modo costante aggiornamenti e che il server risponda a ogni richiesta, ma è direttamente il server che notifica al client il verificarsi di una determinata condizione; il client è comunque responsabile di effettuare nuovamente la richiesta in caso in cui ci siano timeout, ovvero quando il server dopo un certo periodo di tempo non ha ancora inviato una risposta in base ai dati richiesti dal client.

- `Server-Sent Events (SSE)`: sono uno standard definito in HTML5, basato su HTTP, che permette una comunicazione monodirezionale dal publisher verso i subscriber. Gli aggiornamenti vengono inviati in modo automatico, senza bisogno che il client effettui più volte la stessa richiesta.

- `WebSocket`: diventato standard W3C, consente la creazione di canali di comunicazione persistenti e bidirezionali attraverso una singola connessione TCP, garantendo un minor consumo di risorse da parte del subscriber. L'unica correlazione che ha con HTTP è durante la parte di handshake iniziale, in cui le due parti, il client e il server, stabiliscono se possono parlare entrambe con questo protocollo. Poi viene, a tutti gli effetti, aperta una socket e fatto l'upgrade della comunicazione.

Stabilire la tipologia di comunicazione ed eventualmente fare fallback da WebSocket, la migliore, da periodic-polling, la peggiore, non è un compito facile se dovessimo partire da zero. Date anche le premesse precedenti, potrebbe sembrare che realizzare un sistema di comunicazione in real-time sia quasi impossibile, ma in realtà ASP.NET Core include, in una parte dedicata del framework a partire da ASP.NET Core 2.1, un toolkit chiamato SignalR, che va ad astrarre gran parte della complessità, lasciando allo sviluppatore solo una piccola parte di configurazione e di logica, che è dipendente dal processo applicativo. Esattamente come per ASP.NET Core, anche SignalR è stato completamente riscritto da zero ma, nonostante questo, il team di sviluppo ha cercato di mantenere lo stesso approccio delle versioni per ASP.NET, in modo tale che gli sviluppatori non si trovino troppo in difficoltà a migrare una soluzione esistente.

SignalR è una libreria che si divide in due parti, che hanno entrambe lo scopo di fare processing asincrono dei dati in modo rapido: la parte server è in grado di mantenere le connessioni con tutti i consumer, di scalare il protocollo di comunicazione istantaneamente e di codificare i messaggi che deve scambiare, mentre la parte client è un set di librerie che permette la comunicazione con il server e che funziona su qualsiasi piattaforma, compreso il mobile nativo di Android e iOS. In particolare, rispetto alle versioni presenti su ASP.NET, c'è stato un grande passo in avanti per la parte client distribuita per le applicazioni web: costruita in TypeScript e distribuita tramite npm, garantisce maggiori performance, possibilità da parte di Microsoft di mantenerla e aggiornarla con più semplicità e, non meno importante, non ha più la dipendenza

obbligatoria da jQuery, rendendola agnostica rispetto alle librerie JavaScript.

Iniziamo a vedere quanto sia facile lavorare, creando una prima connessione. Poiché, a partire da ASP.NET Core 2.1, SignalR è incluso nel framework, non è necessario aggiungere pacchetti di NuGet, ma è comunque necessario comunicare al server che si vuole fare uso del servizio, configurando, come al solito, il relativo middleware nello startup, come facciamo nell'Esempio 12.12.

Esempio 12.12

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddSignalR();
}
```

All'interno del metodo `Configure` invece, dobbiamo andare a registrare tutti gli `Hub`, ovvero tutte quelle classi che faranno da aggregatori dei messaggi che devono passare tra client e server e viceversa. Un hub, infatti, a livello concettuale non è nient'altro che lo stesso contenitore di azioni, un po' come il controller lo è per le action. La configurazione è contenuta nell'Esempio 12.13.

Esempio 12.13

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
  app.UseSignalR(routes =>
  {
    routes.MapHub<ChatHub>("/chat");
    routes.MapHub<StreamingHub>("/streaming");
  });
}
```

In questo esempio mappiamo su due route ben specifiche altrettanti hub. Le route sono completamente personalizzabili e un ciascun hub è una classe che eredita dalla classe base `Hub` di SignalR, che si comporta da aggregatore e consente di rimandare i messaggi con le varie politiche del caso: a tutti, in broadcast, a un client ben specifico, individuato da

un ConnectionId, oppure a un gruppo specifico di utenti. Nell'Esempio 12.14 viene creato un ChatHub, ovvero un Hub che permette di gestire una chat tra utenti.

Esempio 12.14

```
public class ChatHub : Hub
{
  public void Send(string name, string message)
  {
    Clients.All.SendAsync(″broadcastMessage″, name, message);
  }
  public override Task OnConnectedAsync()
  {
    Clients.All.SendAsync(″broadcastMessage″, ″system″, $″{Context. ConnectionId}
è entrato nella chat.″);
    return base.OnConnectedAsync();
  }
  public override Task OnDisconnectedAsync(Exception exception)
  {
    Clients.All.SendAsync(″broadcastMessage″, ″system″, $″{Context. ConnectionId}
è uscito dalla chat.″);
    return base.OnDisconnectedAsync(exception);
  }
}
```

I metodi OnConnectedAsync e OnDisconnectedAsync vengono utilizzati per comunicare a tutti gli utenti in ascolto, in modalità broadcast, che un determinato utente, identificato da un suo ConnectionId, si è collegato (oppure disconnesso) alla chat corrente. La funzione di Send invece, non è un override di un metodo base esposto dalla classe Hub di SignalR, ma, all'opposto, è una funzione costruita per i nostri scopi: in questo caso il server, così come il client, potrà invocare la funzione Send definita nel ChatHub per notificare a tutti i client che l'utente, definito dalla proprietà name, ha inviato il messaggio message. Saranno poi i vari client a doversi registrare, con una tecnica di tipo publisher-subscribe, all'evento di broadcastMessage, così da essere notificati dell'invio di un messaggio nella chat da parte di qualche altro utente.

Il subscriber che andrà a collegarsi può essere una qualsiasi applicazione desktop, per esempio WPF, piuttosto che un'app mobile, oltre che un'applicazione web. La parte grafica della chat che verrà

realizzata è composta da questa scarsa porzione di HTML, mostrata nell'Esempio 12.15.

Esempio 12.15

```
<body>
  <div class="container">
    <input type="text" id="message" class="message" />
    <input type="button" id="sendmessage" value="Send" class="btn" />
    <ul id="discussion"></ul>
  </div>
  <script type="text/javascript" src="scripts/signalr-client.js"></script>
  <script type="text/javascript" src="scripts/chat.js"></script>
</body>
```

Il codice è costituito da una semplice form – in cui è contenuta una casella di testo, che rappresenta il messaggio da inviare a tutti i client – e dal pulsante `sendmessage`, che chiamerà la funzione `Send` definita lato server per notificare tutti gli altri client. Viene anche aggiunta una lista vuota, chiamata `discussion`, in cui verranno elencati tutti i messaggi ricevuti sotto forma di elenco puntato. La definizione di SignalR lato client è contenuta nel file `signalr-client.js`, scaricato tramite il comando `npm install @aspnet/signalr`.

Allo stesso tempo, va anche definita tutta la logica applicativa, contenuta nel file `chat.js`, esposta nell'Esempio 12.16.

Esempio 12.16

```
document.addEventListener('DOMContentLoaded', function () {
  // recupero il messaggio che voglio inviare
  var messageInput = document.getElementById('message');
  // chiedo all'utente il suo nome (utilizzato nella chat).
  var name = prompt('Inserisci il tuo nome:', '');
  messageInput.focus();
  // apro la connessione sul ChatHub
  startConnection('/chat', function (connection) {
    // creo la funzione 'broadcastMessage' invocata dal 'ChatHub'
    connection.on('broadcastMessage', function (name, message) {
      // aggiungo il messaggio come parte dell'elemento 'discussion'
      var liElement = document.createElement('li');
```

```
      liElement.innerHTML = ´<strong>´ + name + ´</strong>:  ´ +
message;
      document.getElementById(´discussion´).appendChild(liElement);
    });
  }).then(function (connection) {

    console.log(´connection started´);

    // recupero il click del pulsante per inviare il messaggio
    document.getElementById(´sendmessage´).addEventListener(´click´, function
(event) {

      // chiamo il metodo ´Send´ all´interno del ´ChatHub´
      connection.invoke(´send´, name, messageInput.value);

      // resetto i valori di input
      messageInput.value = ´´;
      messageInput.focus();
      event.preventDefault();
    });
  }).catch(error => {
    console.error(error.message);
  });

  function startConnection(url, configureConnection) {
    return function start(transport) {

      console.log(´Starting connection using ${signalR.
TransportType[transport]} transport´);

      var connection = new signalR.HubConnection(url, { transport: transport });

      if (configureConnection && typeof configureConnection === ´function´) {
        configureConnection(connection);
      }
      return connection.start()
        .then(function () {
          return connection;
      })
        .catch(function (error) {
          console.log(´Cannot start the connection use ${signalR.
TransportType[transport]} transport. ${error.message}´);

          if (transport !== signalR.TransportType.LongPolling) {
            return start(transport + 1);
          }

          return Promise.reject(error);
        });
    }(signalR.TransportType.WebSockets);
  }
});
```

La logica del codice JavaScript consiste nell'aspettare che venga caricato il DOM della pagina, quindi viene chiesto all'utente, tramite una dialog, il suo nome: questo valore verrà salvato per essere poi rimandato nella proprietà `name` al server. Viene quindi recuperato il messaggio e viene avviata la connessione sullo hub `ChatHub`, che può avvenire in diverse modalità: la prima, la più diffusa tra i browser moderni, è basata sul protocollo WebSocket ma, come abbiano anticipato, in caso di non

funzionamento perché non supportata, la logica è in grado di fare fallback su Server-Sent Events (SSE) e long polling, prima di rifiutare la connessione, poiché SignalR è in grado di fare protocol negotiation con il client.

Una volta ottenuto un collegamento verso il server con una delle modalità appena descritte, diventa fondamentale registrarsi all'evento di click del pulsante della form: a ogni pressione, infatti, verrà recuperato il messaggio scritto dall'utente, che verrà rimandato, assieme al nome dell'utente, alla funzione `Send` definita lato server tramite una chiamata a `invoke` dell'oggetto `connection` ottenuto in precedenza. Poiché lato server il messaggio viene inviato a tutti i client in modo indifferente senza escludere nessuno, anche il sender stesso del messaggio lo riceverà. Per questo, grazie alla funzione `on` richiamata sull'oggetto `connection`, ci possiamo mettere in ascolto di un messaggio inviato dalla funzione `broadcastMessage` chiamata dal server. Alla ricezione dei dati, verrà semplicemente aggiunto un nuovo punto all'elenco puntato definito sulla form come `discussion`.

Una delle novità di SignalR per ASP.NET Core rispetto al passato è il supporto per lo streaming: grazie alle performance raggiunte con la nuova infrastruttura, è possibile inviare dati da server a client prima dell'invocazione sull'hub, con un numero decisamente maggiore di client rispetto al passato, che può essere collegato contemporaneamente. Vediamo ora un esempio di streaming, definendo uno hub come quello dell'Esempio 12.17.

Esempio 12.17

```
public class StreamingHub : Hub
{
  public IObservable<string> StartStreaming()
  {
    return Observable.Create(
      async (IObserver<string> observer) =>
      {
        for (var i = 0; ; i++)
        {
          observer.OnNext($"Invio messaggio {i}...");
          await Task.Delay(1000);
        }
      });
  }
```

```
}
```

Al contrario del `ChatHub` mostrato in precedenza, seguendo questa nuova tecnica non è più importante registrarsi agli eventi di connessione e disconnessione del client, perché abbiamo solo un metodo chiamato `StartStreaming` che apre, attraverso l'uso di `System.Reactive.Linq`, un oggetto di tipo `Observable`. Il funzionamento consiste solamente nell'invio di un messaggio su tutti i subscriber registrati all'oggetto `Observable`, all'infinito e con un ritardo temporale tra l'invio di un messaggio e il successivo di circa un secondo.

Il client, che per fare una variazione sul tema sarà una console application .NET Core, avrà solamente la necessità di aggiungere una reference al relativo pacchetto di NuGet `Microsoft.AspNetCore.SignalR.Client` e implementare poche righe di codice per leggere i dati in tempo reale. L'Esempio 12.18 contiene il codice necessario a implementare questa funzionalità.

Esempio 12.18

```
static async Task MainAsync(string[] args)
{
  var streamingConnection = new HubConnectionBuilder()
                .WithUrl("http://localhost:5000/streaming")
                .WithConsoleLogger()
                .Build();

  // apertura della connessione
  await streamingConnection.StartAsync();

  // apertura del canale per lo streaming sulla funzione StartStreaming
  var channel = await _streamingConnection.StreamAsChannelAsync<string>("StartS
treaming");

  // si aspetta l´arrivo di un messaggio
  while (await channel.WaitToReadAsync())
  {
    // lettura del messaggio e stampa su console
    while (channel.TryRead(out string message)
      Console.WriteLine($"Messaggio ricevuto: {message}");
  }
}
```

La prima cosa fatta è stata creare un oggetto di tipo `HubConnectionBuilder` per aprire la connessione verso la route dov'è definito lo `StreamingHub` lato server, per poi avviare la connessione con

la chiamata al metodo asincrono `StartAsync` e, infine, abbiamo creato l'oggetto `channel` sul quale ascoltiamo messaggi provenienti dalla funzione `StartStreaming` definita lato server. Ogni messaggio inviato dal server verrà letto dal client in modalità asincrona e quindi scritto sulla console. Tutta la complessità definita con le precedenti versioni di SignalR si è ridotta all'osso e, comparata con altri framework analoghi, questa opzione garantisce sicuramente ottime prestazioni nonostante sia un prodotto relativamente nuovo e costruito da zero.

Quello che è dato per scontato e di cui non abbiamo ancora discusso riguarda come i dati vengono serializzati per poter essere inviati come messaggio dal publisher al subscriber e viceversa: di default c'è il supporto al classico text-based con messaggi in JSON, ma SignalR per ASP.NET Core introduce anche il supporto per MessagePack, che effettua la serializzazione in formato binario, in modo da garantire messaggi di dimensioni inferiori e più veloci da spedire. MessagePack non è integrato nel framework ma è distribuito come pacchetto di NuGet `Microsoft.AspNetCore.SignalR.MsgPack`. L'aggiunta di questo formato è illustrata nell'Esempio 12.19.

Esempio 12.19

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddSignalR()
          .AddMessagePackProtocol();
}
```

Per la parte client, invece, è necessario aggiungere il pacchetto di NuGet `Microsoft.AspNetCore.SignalR.Client.MsgPack` e specificare il protocollo di serializzazione in fase di collegamento all'hub, come è illustrato nell'Esempio 12.20.

Esempio 12.20

```
_var streamingConnection = new HubConnectionBuilder()
                .WithUrl("http://localhost:5000/streaming")
                .WithMessagePackProtocol()
                .WithConsoleLogger()
```

```
                .Build();
```

Per maggiori informazioni sul funzionamento del protocollo MessagePack e per capire se è supportato dalla nostra architettura, rimandiamo alla documentazione ufficiale disponibile all'indirizzo: http://aspit.co/bnn.

Conclusioni

All'interno di questo capitolo sono stati affrontati dei temi avanzati rispetto alla creazione di una WebAPI classica e, in particolare, si è parlato di tutto ciò che può essere utile per riuscire a mantenere il servizio nel tempo, trattando in particolare gli scenari relativi alla documentazione con Swagger e al versionamento. Proseguendo all'interno del capitolo, si è visto come fare uso dei WebHook, ovvero di una sorta di callback HTTP, per estendere le capacità delle API distribuite su più livelli, in modo che possano lavorare ed evolvere indipendenti gli uni dagli altri, garantendo in un qualche modo la continuità di servizio. Al contrario, invece, abbiamo visto che esistono sistemi che hanno forti necessità di comunicare in tempo reale, e per questi non c'è modo di avere una separazione tra il client e il server, poiché entrambi devono potersi "conoscere" e "parlare" nel più breve tempo possibile e con messaggi che siano i più piccoli possibili, pertanto si è discusso della nuova versione di SignalR, integrata in ASP.NET Core, e di come vada a implementare, rispetto al passato, un paradigma decisamente più moderno e vicino agli sviluppatori, che nella sua ultima versione raggiunge performance notevoli, garantendo la possibilità di lavorare in streaming, ma consentendo anche l'indipendenza da pacchetti esterni come jQuery.

Negli esempi affrontati nel corso del capitolo, l'ambiente era controllato, pertanto la generazione di qualche eccezione era principalmente voluta ma, nell'ambiente di produzione, quando si rilasciano questa ed altre funzionalità, non è proprio quello che ci si aspetta. Per questo è necessario iniziare a parlare, come vedremo nel prossimo capitolo, di strumenti legati alla diagnostica e alle performance, per capire nel più breve tempo possibile cosa eventualmente sta

generando (o genererà) problemi nelle nostre applicazioni, per essere in grado di intervenire e risolverli.

13

Gestione e diagnostica degli errori

I capitoli precedenti sono sufficienti per capire e sviluppare un applicativo completo che sappia gestire form o fornire servizi REST, ma una soluzione ben fatta deve saper gestire anche le situazioni impreviste, dare messaggi di errore amichevoli agli utenti e raccogliere informazioni utili a una eventuale diagnostica. In caso di errore dobbiamo accorgercene o, in caso di malfunzionamento segnalato dall'utente, dobbiamo poter ricostruire la situazione che si è verificata, capire lo stato dei dati o identificare logiche errate che non necessariamente causano un errore bloccante all'utente.

In questo capitolo vogliamo mostrare tutti gli strumenti inclusi in ASP.NET Core per poter gestire gli errori, tracciare tutto ciò che avviene ed eventualmente appoggiarci a servizi terzi, come quelli forniti da Microsoft Azure, per avere statistiche o capire l'impatto di un errore.

La gestione degli errori a runtime

Quando parliamo di errore, stiamo in realtà generalizzando, perché in un applicativo gli errori possono essere di natura diversa, con livelli diversi. A fronte di una richiesta fatta dall'utente, la risposta che può giungere all'utente può essere un messaggio di errore generico, uguale per tutte le pagine, oppure mirato a fornire informazioni sull'operazione appena fatta dall'utente, ma che non ha raggiunto gli scopi prefissati. Ne è un esempio la validazione della form vista nel Capitolo 7: la pagina appena inviata viene mostrata nuovamente all'utente, ma fornendo messaggi di errore su ogni singolo campo o sull'intera form. Questo genere di comportamento è da preferire in ogni situazione, perciò in ogni pagina e

azione dobbiamo sempre domandarci cosa potrebbe andare storto: il codice contempla variabili nulle o vuote? Vengono gestite possibili eccezioni? Quest'ultima domanda è sempre la più difficile alla quale rispondere, ma non c'è dubbio che, in scenari cloud e sempre più distribuiti, una chiamata a un database o a un servizio web può, in alcuni casi, andare in errore, e questa situazione va prevista.

Nei nostri controller e nelle nostre pagine è opportuno quindi utilizzare il costrutto `try/catch` per intercettare questo genere di eccezioni e mostrare un messaggio di errore amichevole all'utente. È inutile, infatti, dare dettagli tecnici, ed è da preferire l'uso di messaggi più generici.

Il `ModelState`, utilizzato per popolare gli errori del modello, è un buon contenitore per inserire anche errori generici, come viene mostrato nell'Esempio 13.1.

Esempio 13.1

```
public void OnPost()
{
  try
  {
    // Codice che chiama dei servizi...
  }
  catch (WebException)
  {
    // Aggiungo un errore generico al modello
    ModelState.AddModelError(″″,
      ″Errore nella chiamata ai servizi, riprovare.″);
  }
}
```

Il primo parametro della funzione `AddModelError` indica il nome della proprietà del modello alla quale associare l'errore. Se non specificato, l'errore è generico e può essere facilmente visualizzato nella view attraverso il tag helper, come è mostrato nell'Esempio 13.2.

Esempio 13.2

```
<form asp-page=″Index″>
  <div asp-validation-summary=″ModelOnly″></div>
```

L'utilizzo dell'enumerato `ModelOnly` forza la visualizzazione di errori generici del modello ed è facoltativo.

Gli errori durante lo sviluppo

Per quanto riguarda, invece, gli altri tipi di errore, cioè quelli che non prevediamo o che sono frutto di codice non scritto correttamente, possiamo ricorrere a una gestione generica che copra l'intero applicativo. ASP.NET Core è in grado di intercettare eccezioni generate all'interno della richiesta e, come prevede il protocollo HTTP, risponde con uno status code 500 visualizzato con una pagina generica di errore del browser, di poca utilità per l'utente finale, anche in caso si richieste REST.

Abbiamo quindi già visto nel Capitolo 3 che il template di Visual Studio predispone la classe `Startup` con la configurazione di due middleware, come viene mostrato nell'Esempio 13.3.

Esempio 13.3

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
  if (env.IsDevelopment())
  {
    app.UseDeveloperExceptionPage();
  }
  else
  {
    app.UseExceptionHandler("/Error");
  }
```

I due middleware differenziano il comportamento da adottare nel caso in cui ci troviamo nell'ambiente di sviluppo o di produzione. `UseDeveloperExceptionPage` installa un middleware che intercetta le eccezioni generate e produce un HTML utile a noi sviluppatori per diagnosticare l'errore. Questo middleware non va abilitato in scenari di produzione perché, oltre a non fornire un'interfaccia consona all'utente, può dare informazioni sensibili o utili a un eventuale hacker che vuole capirne il funzionamento.

L'extension method contiene un overload al quale è possibile passare un oggetto di tipo `DeveloperExceptionPageOptions` che dispone di due

proprietà: `SourceCodeLineCount` e `FileProvider`. Il primo ci permette di indicare quante righe della sorgente possiamo visualizzare, mentre il secondo ci permette di personalizzare i provider da utilizzare per leggere il file sorgente, necessario solo se il codice non è nostro.

Un secondo middleware, utile sempre per la fase di sviluppo, è configurabile tramite l'extension method `UseDatabaseErrorPage`, come viene mostrato nell'Esempio 13.4.

Esempio 13.4

```
if (env.IsDevelopment())
{
  app.UseDeveloperExceptionPage();
  app.UseDatabaseErrorPage();
}
```

Intercetta le eccezioni relative a Entity Framework Core e, in particolare, quelle riguardanti la migrazione della struttura del database. Qualora siano pendenti una o più migrazioni, una pagina web suggerisce l'operazione da fare, come viene mostrato nella Figura 13.1.

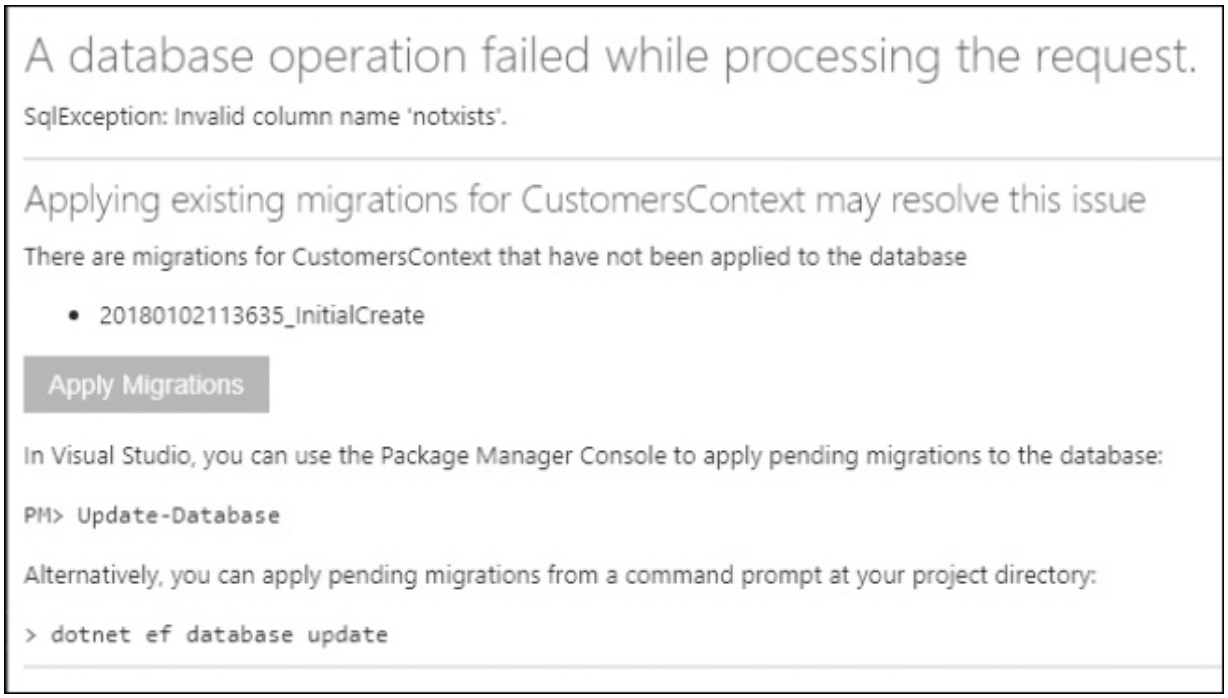


Figura 13.1 – Pagina di errore in caso di migrazione pendente di Entity Framework.

Premendo il pulsante non facciamo altro che attivare la procedura di update, la stessa che sarebbe possibile avviare via codice o utilizzando DotNet CLI. Questo middleware va di conseguenza adottato se usiamo il meccanismo delle migrazioni e solo per facilitare la loro applicazione. Inoltre, è di fondamentale importanza la configurazione del middleware fatta nell'Esempio 13.4, effettuata successivamente a quella di gestione generica degli errori.

Gli errori per l'utente

Riprendiamo ora l'Esempio 13.3 e la chiamata al metodo `UseExceptionHandler`, un ulteriore middleware da attivare per gli scenari di produzione. Esso intercetta tutte le eccezioni generate nel nostro codice server e fornisce una pagina di errore standard per gestirle. Nel template generato da Visual Studio, troviamo l'uso del percorso `/Error`, una pagina implementata attraverso una action di MVC o una Razor Page, a seconda della tipologia di progetto scelto.

> *Questi extension method per la gestione degli errori devono essere chiamati prima di tutti gli altri dedicati alla configurazione dei middleware. Fungono da contenitori per la restante parte della pipeline e ne intercettano eventuali errori.*

Il suo aspetto predefinito è visibile nella Figura 13.2.

Error.

An error occurred while processing your request.

Request ID: `|cbddc7ec-4553af7aa12cde39.`

Development Mode

Swapping to **Development** environment will display more detailed information about the error that occurred.

Development environment should not be enabled in deployed applications, as it can result in sensitive information from exceptions being displayed to end users. For local debugging, development environment can be enabled by setting the **ASPNETCORE_ENVIRONMENT** environment variable to **Development**, and restarting the application.

Figura 13.2 – Pagina di errore standard per l'utente.

Tra le cose da fare prima della messa in produzione, c'è sicuramente la personalizzazione di questa pagina per mostrare un messaggio di errore nella lingua corretta e senza alcuni aspetti tecnici che non riguardano l'utente. Inoltre, possiamo scegliere di cambiare il percorso della pagina a seconda delle nostre esigenze.

Oltre a intervenire sull'HTML della view, possiamo modificare il modello che sfruttiamo per dare informazioni all'utente, simile all'Esempio 13.5.

Esempio 13.5

```
public class ErrorModel : PageModel
{
  public string RequestId { get; set; }
  public bool ShowRequestId => !string.IsNullOrEmpty(RequestId);
  public void OnGet()
  {
    RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier;
  }
}
```

La proprietà `RequestId` viene messa a disposizione per mostrare l'identificativo della richiesta corrente, ottenuta dal sistema di diagnostica di .NET o l'identificativo fornito dall'infrastruttura di ASP.NET

Core. Questo codice, come vedremo più avanti nel capitolo, può essere mostrato all'utente per consentirgli eventualmente di effettuare segnalazioni e con esso darci la possibilità di correlare un eventuale logging all'errore che si è verificato, metodo decisamente più efficace rispetto a identificare il contesto di nostro interesse ricercandolo in un intervallo temporale.

La personalizzazione può avvenire anche sfruttando alcune informazioni aggiuntive che vengono aggiunte nel contesto e, in particolare, l'eccezione che ha portato alla visualizzazione della pagina di errore. Attraverso la collezione `Features` dell'oggetto `HttpContext` possiamo accedere a quella di tipo `IExceptionHandlerFeature` contenente la proprietà `Error`. Come riportato nell'Esempio 13.6, sfruttiamo questa informazione per cambiare il messaggio da mostrare all'utente.

Esempio 13.6

```
public string ErrorMessage
{
  get
  {
    // Accedo alla feature
    var exceptionHandlerFeature =
      HttpContext.Features.Get<IExceptionHandlerFeature>();

    // Personalizzo il messaggio da mostrare
    switch (exceptionHandlerFeature?.Error)
    {
      case WebException we:
        return "Errore nella chiamata ai servizi. Riprovare";
      default:
        return "Si è verificato un errore non previsto. Riprovare";
    }
  }
}
```

Questa proprietà può essere poi visualizzata nella view attraverso Razor, per fornire un'interfaccia più ricca.

Dobbiamo sottolineare che quando si verifica un errore e la pagina specifica viene visualizzata, l'indirizzo della pagina è ancora quello che ha generato l'eccezione. L'utente non subisce nessun redirect HTTP, ma

vengono processate due pagine: la prima, che generato il problema, e la seconda, che mostra il messaggio.

Possiamo in alternativa usare l'overload dell'extension method `UseExceptionHandler` che accetta una funzione in grado di configurare un `IApplicationBuilder`, un po' come già facciamo sulla funzione `Configure` dello `Startup`. L'unica differenza è che in essa dobbiamo configurare i middleware da richiamare in caso di errore. Nell'Esempio 13.7 sfruttiamo questa possibilità tramite un middleware basato su lambda per rimandare l'utente a una pagina di errore, questa volta però tramite redirect HTTP.

Esempio 13.7

```
app.UseExceptionHandler(b => b.Use((context, next) =>
{
  context.Response.Redirect("/Error");
  // Nessuna operazione asincrona
  return Task.CompletedTask;
}));
```

Così facendo abbiamo il pieno controllo sulla risposta da dare all'utente anche se tramite il redirect perdiamo l'informazione sull'errore accessibile tramite feature, poiché appartenente alla richiesta che ha generato l'errore e non alla seconda dove l'utente è stato rimandato. Questo overload può essere utile anche nel caso in cui vogliamo fornire una risposta JSON a fronte di una richiesta REST, per la quale una risposta HTML non sarebbe adeguata. Gli errori delle nostre pagine e gli status code 500, però, non sono gli unici dei quali ci dobbiamo preoccupare.

Le pagine per gli status code

Tra gli status code più comuni in cui l'utente si può imbattere c'è sicuramente il 400, per una richiesta errata, un 404 per una pagina non trovata o un 403 per un accesso vietato. Questi codici, che vanno dal 400 al 599, possono essere generati da qualsiasi middleware: da una action di MVC, da una funzione di una Razor Page, da una web API o da un file

statico. Quando si verificano, anche in questo caso, il browser mostra una pagina bianca.

A questo scopo vengono in aiuto altri middleware. Il più semplice si configura con l'extension method `UseStatusCodePages` il quale, a fronte di una risposta con i codici prima citati, produce il semplice HTML visibile nella Figura 13.3.

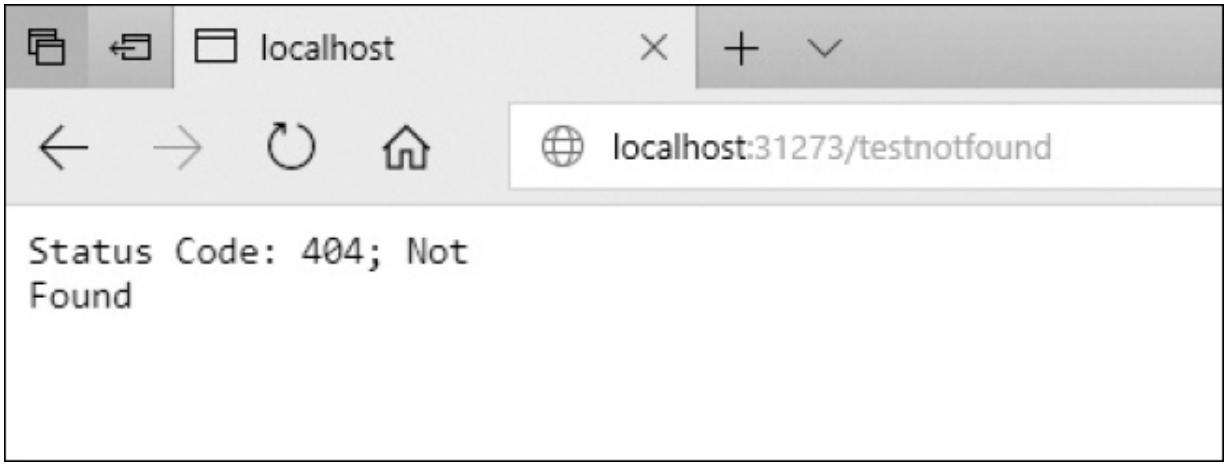


Figura 13.3 – Pagina predefinita per lo status code 404.

Possiamo configurare il middleware affinché utilizzi una nostra funzione o un nostro middleware, in maniera del tutto identica a quanto visto nell'Esempio 13.7, ma il metodo migliore per produrre pagine HTML di più elevata qualità è sicuramente quello di affidarsi ad altri due middleware: `UseStatusCodePagesWithRedirects` e `UseStatusCodePagesWithReExecute`. Il primo effettua un redirect HTTP, mentre il secondo una esecuzione all'interno della richiesta stessa, come fa `UseExceptionHandler`. Utilizziamo quindi quest'ultimo per realizzare una view tramite Razor Page, come mostrato nell'Esempio 13.8.

Esempio 13.8

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
  // Gestione errori
  app.UseExceptionHandler("/error")
  // Personalizzazione degli status code
```

```
app.UseStatusCodePagesWithReExecute("/status/{0}");
// Altro...
app.UseStaticFiles();
```

L'utilizzo del placeholder "{0}" ci permette di inserire dinamicamente il codice all'interno del percorso e quindi implementare in un unico punto la pagina di stato.

Possiamo implementare questa semplice pagina, per esempio, con una Razor Page con una regola di routing personalizzata, in modo da ricevere lo status code nel percorso indicato, come viene mostrato nell'Esempio 13.9.

Esempio 13.9

```
@page "{code}"
@using Microsoft.AspNetCore.Routing
@{
  string code = HttpContext.GetRouteValue("code").ToString();
  ViewBag.Title = code;
}
<style>
  .big {
    font-size: 200pt;
    line-height: 0;
    color: #ccc
  }
</style>
<h1 class="big">@code</h1>
```

Nella view è importante notare il placeholder di nome `code` e la lettura del suo valore a runtime. Il codice è piuttosto semplice, perciò non abbiamo definito un modello apposito ma tutta la logica è stata definita nella view stessa, il cui risultato è visibile nella Figura 13.4.

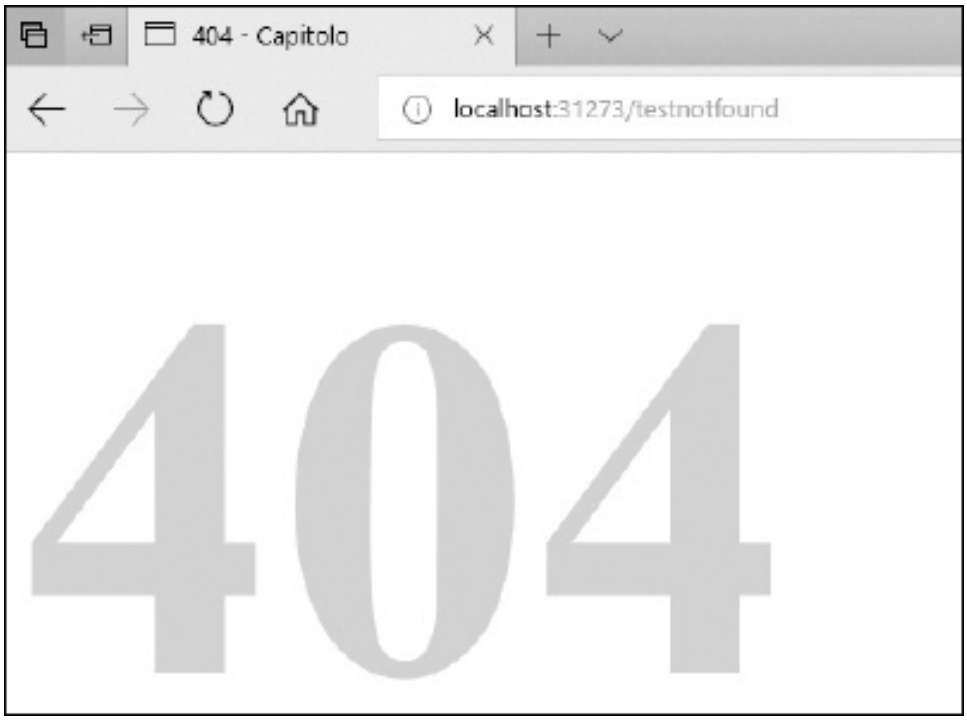


Figura 13.4 – Pagina personalizzata per lo status code 404.

Vi sono, infine, delle situazioni in cui non vogliamo che il middleware in questione personalizzi la risposta ma vogliamo forzare il comportamento predefinito. Anche in questo caso viene in aiuto una feature di nome `IStatusCodePagesFeature`, che tramite la sua proprietà `Enabled` ci permette di disabilitare per la richiesta corrente il middleware della status page, come mostrato nell'Esempio 13.10.

Esempio 13.10

```
public IActionResult TestNotFound()
{
    // Disattivo il middleware per gli status page
    var statusCodePagesFeature = HttpContext.Features
      .Get<IStatusCodePagesFeature>();
    statusCodePagesFeature.Enabled = false;

    return NotFound();
}
```

Nell'esempio viene disattivata l'opzione durante una action di un controller, prima che restituisca un 404. Dopo aver visto come gestire i

messaggi di errore e le informazioni nei confronti dell'utente, vediamo ora come possiamo raccogliere informazioni utili alla diagnostica.

Tracciare gli eventi

Nel Capitolo 3 abbiamo introdotto i due servizi `ILogger` e `ILoggerFactory`, con i quali possiamo tracciare eventi all'interno del nostro applicativo. Tramite la dependency injection, possiamo sfruttare questi oggetti all'interno di tutti i layer applicativi, dal presentation costituito da ASP.NET Core, fino ad arrivare all'accesso ai dati o a un canale di comunicazione. Il namespace `Microsoft.Extensions.Logging` e l'omonimo pacchetto NuGet compatibile .NET Standard rendono questi componenti indipendenti e utilizzabili anche su altri runtime.

Attraverso un'istanza di `ILogger` possiamo tracciare qualsiasi stringa di testo a noi utile per capire cosa è successo, perciò è fondamentale essere più descrittivi e ricchi nelle informazioni fornite. Riprendiamo un esempio su come tracciare un'informazione durante una chiamata a una action di un controller MVC.

Esempio 13.11

```
public class LoggingController : Controller
{
  private readonly ILogger<LoggingController> _logger;
  public LoggingController(ILogger<LoggingController> logger)
  {
    _logger = logger;
  }
  public IActionResult Index(string mode, int type)
  {
    _logger.LogInformation("Index with {modeValue} and {typeValue}",
    mode, type);
  return View();
}
```

La funzione `LogInformation` accetta un messaggio e, facoltativamente, uno o più parametri. Diversamente da quanto potremmo pensare, il messaggio supporta una sintassi di template ma non usa la string interpolation di C#. I segnaposto messi tra graffe di nome `modeValue` e

`typeValue` indicano i nomi dei segnaposto stessi, mentre il loro rispettivo valore viene assegnato in modo posizionale con i parametri `mode` e `type`, passati alla funzione. È di fondamentale importanza, quindi, l'ordine dei segnaposto e dei parametri passati. Diversamente dai tradizionali sistemi di log, il tracciamento viene effettuato in modo strutturato, mantenendo separati il messaggio e i parametri. Spetta poi al provider che lo mostra o lo persiste se decidere di formattare il template o di memorizzare separatamente i valori.

> *Poiché il tracciamento viene effettuato in maniera strutturata per poi essere memorizzato, è opportuno dosare i parametri, che possono essere anche oggetti complessi, per evitare di avere database spropositati con informazioni superflue.*

Questo modo di tracciare migliora la qualità dell'informazione e la sua ricerca, decisamente più accurata rispetto a una semplice analisi testuale.

Sebbene più informazioni tracciamo meglio è, ci è facile cadere nella sovrabbondanza di dati che finiscono poi per creare più confusione e rendere più complicata la diagnostica di un problema. Per questo motivo, ogni tracciamento che facciamo viene accompagnato da un livello e per ognuno di essi disponiamo di un extension method per facilitarci il loro utilizzo. Nell'Esempio 13.12 possiamo vederli all'opera, in ordine d'importanza.

Esempio 13.12

```
_logger.LogTrace(″Linea dettagliata con potenziali informazioni sensibili″);
_logger.LogDebug(1001, ″Informazioni di debug, utili per diagnostica″);
_logger.LogInformation(″Dati generici e descrittivi″);
_logger.LogWarning(″Avviso non bloccante″);
_logger.LogError(5001, exception, ″Si è verificato un errore, bloccante″);
_logger.LogCritical(exception, ″Errore catastrofico, perdita di dati″);
```

Tutti gli extension method dell'Esempio 13.12 dispongono di vari overload che permettono di specificare template, parametri e anche

l'oggetto `Exception`, utile soprattutto quando utilizziamo le funzioni `LogError` e `LogCritical`. Facoltativamente, il primo parametro di ogni funzione accetta un identificativo, un intero che ci permette di identificare il tracciamento più precisamente di una stringa. Il livello, nell'Esempio 13.12, utilizzato in ordine crescente, ci permette di scremare il dettaglio di informazioni da mostrare o da persistere. È di fondamentale importanza tracciare le informazioni utilizzando il livello più appropriato e non cadere nella pigrizia che spesso porta a utilizzare uniformemente il livello `Information`.

I livelli hanno un'importanza che non è solo formale ma anche pratica. Ogni provider può decidere autonomamente quali livelli onorare, dando un livello minimo. Se, per esempio, il debug ha un livello minimo su `Warning`, vengono memorizzati eventi solo di livello uguale o superiore, quindi: `Warning`, `Error` e `Critical`. Il provider dedicato alla console, produce l'output della Figura 13.5 con il suo comportamento predefinito. Quando utilizziamo IIS Express, esso è visibile attraverso una finestra apposita di Visual Studio.

```
Output
Show output from: ASP.NET Core Web Server
Capitolo13> info: Capitolo13.Controllers.LoggingController[0]
Capitolo13>       Index with my and 2
Capitolo13> dbug: Capitolo13.Controllers.LoggingController[1001]
Capitolo13>       Informazioni di debug, utili per diagnostica
Capitolo13> info: Capitolo13.Controllers.LoggingController[0]
Capitolo13>       Dati generici e descrittivi
Capitolo13> warn: Capitolo13.Controllers.LoggingController[0]
Capitolo13>       Avviso non bloccante
Capitolo13> fail: Capitolo13.Controllers.LoggingController[5001]
Capitolo13>       Si è verificato un errore, bloccante
Capitolo13> System.Exception: test
Capitolo13> crit: Capitolo13.Controllers.LoggingController[0]
Capitolo13>       Errore catastrofico, perdita di dati
Capitolo13> System.Exception: test
Capitolo13> info: Microsoft.AspNetCore.Mvc.StatusCodeResult[1]
Capitolo13>       Executing HttpStatusCodeResult, setting HTTP status code 200
```

Figura 13.5 – Output in console attraverso Visual Studio.

Nella figura possiamo vedere una formattazione del testo che mostra il livello abbreviato, il messaggio formattato e, tra parentesi quadre, i nostri identificativi 1001 e 5001 utilizzati e, laddove non specificati,

assunti con valore a 0. La categoria, dedotta dal tipo T di `ILogger` richiesto nell'Esempio 13.1, ricopre un ruolo importante, perché rappresenta un ulteriore strumento per differenziare e ricercare le informazioni tracciate. Nella Figura 13.5, all'ultima riga, possiamo vedere che non siamo gli unici a sfruttare questo strumento. L'intero ASP.NET Core è scritto affinché anch'esso tracci, con categorie sotto il namespace `Microsoft`, livelli e identificativi differenti.

Con il tracciamento abbiamo anche la possibilità di circoscrivere in uno scope più messaggi all'interno di un unico contesto, anch'esso identificato dal messaggio e da eventuali parametri. Nell'Esempio 13.13 vediamo come aprire uno scope e come chiuderlo grazie al pattern dispose.

Esempio 13.13

```
// Inizio di un nuovo scope con parametri
using (_logger.BeginScope("Nuovo scope {mode}", mode))
{
  _logger.LogInformation("Dati generici e descrittivi");
  _logger.LogWarning("Avviso non bloccante");
}
```

Gli scope non sono normalmente visualizzati in console, perché amplificano la quantità di informazioni mostrate, ma è possibile agire su essa tramite l'opzione `IncludeScopes` in fase di configurazione del provider.

Configurare il logging

Quando creiamo un nuovo progetto ASP.NET Core, godiamo automaticamente di una serie di configurazioni implicite, che ci permettono di partire immediatamente con lo sviluppo. Il metodo `CreateDefaultBuilder` che troviamo nel `Program.cs` effettua una serie di configurazioni dedicate al logging, che possiamo trovare ricostruite nell'Esempio 13.14.

Esempio 13.14

```
webhost.
  .ConfigureLogging((h, l) =>
  {
    l.AddConfiguration(h.Configuration.GetSection(″Logging″));
    l.AddConsole();
    l.AddDebug();
   })
```

La funzione di configurazione `ConfigureLogging` permette di passare un delegato che accetta il contesto di configurazione (parametro h) e un `ILoggingBuilder` (parametro l) che, nello stile di .NET Core, permette di aggiungere uno o più provider. La chiamata a `AddConfiguration` aggancia la sezione `Logging` della configurazione, per consentirci di personalizzare alcuni aspetti di logging senza dover ricompilare il nostro codice. Le chiamate `AddConsole` e `AddDebug` aggiungono rispettivamente i provider che scrivono in console e nella finestra di debug di Visual Studio.

In uno scenario reale, quel codice è implicitamente presente e a noi resta solo il compito di una personalizzazione di `Program.cs`, sfruttando, per esempio, altri provider che troviamo implementati.

Esempio 13.15

```
public static IWebHost BuildWebHost(string[] args) =>
  WebHost.CreateDefaultBuilder(args)
    .ConfigureLogging(l =>
    {
      // Aggiungo i provider
      l.AddEventSourceLogger();
      l.AddTraceSource(″mySwitch″);
      l.AddProvider(myLoggerProviderInstance);
    // Imposto il livello minimo
    l.SetMinimumLevel(LogLevel.Debug);
    })
    .UseStartup<Startup>()
    .Build();
```

L'extension method `AddEventSourceLogger` configura l'utilizzo del registro di sistema per la memorizzazione dei propri tracciamenti, mentre `AddTraceSource` configura la scrittura su

System.Diagnostics.Trace, il precedente sistema di tracciamento nato con il .NET Framework. Infine, la chiamata a AddProvider permette di aggiungere un'istanza personalizzata di provider, anche se nella maggior parte dei casi sono sufficienti gli extension method di configurazione. Cercando all'interno di NuGet, è facile trovare altri provider che, una volta installati, vi mettono a disposizione un nuovo extension method con il prefisso Add da chiamare.

Nell'Esempio 13.15 troviamo, infine, una chiamata a SetMinimumLevel, con il quale possiamo forzare il livello minimo che ogni provider riceve. Così facendo, il motore va automaticamente a escludere tracciamenti di livello inferiore, in maniera del tutto trasparente al provider.

Non solo: in alternativa all'istruzione via codice, possiamo sfruttare l'aggancio fatto alla configurazione nell'Esempio 13.14, per gestire questo comportamento direttamente dal file appsettings.json o tramite altri provider, a seconda di come il sistema delle opzioni è stato impostato. Quando creiamo un progetto direttamente con Visual Studio, troviamo due file: appsettings.json e appsettings.Development.json. Il secondo, contenente la configurazione letta per le fasi di sviluppo, contiene il JSON dell'Esempio 13.16.

Esempio 13.16

```
{
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  }
}
```

La sezione Logging contiene un nodo LogLevel e una chiave Default con la quale viene configurato il livello minimo da adottare, allo stesso modo di quanto fatto via codice nell'Esempio 13.15. Non solo: per ogni categoria o prefisso di categoria, vengono impostati livelli diversi. Quello che otteniamo è la visualizzazione in console di tutti gli eventi per

qualsiasi livello, mentre per quelli generati da `Microsoft`, il livello dev'essere uguale o superiore a `Information`. La Figura 13.5, che abbiamo già visto, lo dimostra e, se necessario, possiamo abilitare il livello `Debug` o `Trace` anche sui tracciamenti di ASP.NET Core. Possiamo inoltre aggiungere una o più voci dedicate alle nostre categorie, circoscrivendole a `Capitolo13` o `Capitolo13.Controllers`.

Le possibilità non sono finite, perché possiamo configurare il livello da adottare anche per uno specifico provider o per una specifica categoria di un certo provider, come mostrato nell'Esempio 13.17.

Esempio 13.17

```
{
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information",
      "Capitolo13": "Trace"
    },
    "Console": {
      "LogLevel": {
        "Microsoft.AspNetCore.Mvc.Razor.Razor": "Debug",
        "Microsoft": "Error",
        "Default": "Information"
      }
    }
  }
}
```

Una volta identificato il provider sul quale vogliamo intervenire, la sintassi da usare è la medesima vista in precedenza: prefisso o nome completo della categoria, e `Default` per tutte le altre categorie. Dobbiamo sottolineare l'ordine di precedenza delle impostazioni fatte, poiché vince prima di tutto la categoria più specifica, mentre a parità di categoria vince l'ultima definita. In entrambi i casi, è indifferente se la regola è stata impostata a livello di provider o in modo generico, come è stato fatto nell'Esempio 13.16. Interessante, infine, è sapere che le modifiche alla configurazione vengono applicate immediatamente senza necessità di riavviare l'host.

Quanto è stato fatto da configurazione è possibile farlo anche da codice, se per caso volessimo godere della massima libertà di espressione, come viene mostrato nell'Esempio 13.18.

Esempio 13.18

```
.ConfigureLogging(l =>
{
  l.AddFilter((provider, category, logLevel) =>
  {
    // Solo i nostri eventi informativi
    if (logLevel == LogLevel.Information && category.StartsWith("Capitolo13"))
    {
      return true;
    }
    return false;
  });
})
```

Il metodo `AddFilter` dispone di molti overload per scegliere quali parametri o a quale categoria ci rivolgiamo ma, nell'Esempio 13.18, utilizziamo quello più versatile perché fornisce tutte le informazioni necessarie alla nostra logica e ci permette di indicare se tracciare (restituendo true) oppure no. È importante tenere presente che questo filtro viene chiamato solo se non abbiamo specificato un livello predefinito nella configurazione e solo per le categorie orfane di livello minimo. Visti tutti gli elementi che compongono il logging, non ci resta che vedere come implementare un provider personalizzato.

Un provider per il logging personalizzato

L'architettura estremamente modulare di ASP.NET Core ci permette anche nel logging di fornire un provider personalizzato, al pari degli altri già implementati. `Console` e `Debug` sono spesso insufficienti e può rendersi necessario l'invio di informazioni a sistemi di tracciamento esterni, database, sistemi di comunicazione o anche al semplice invio di un'e-mail al verificarsi di un errore. Un provider è rappresentato dall'interfaccia `ILoggerProvider` che occorre registrare nel container, come tutte le altre dipendenze. In linea con il pattern di configurazione

dell'intero .NET Core, possiamo quindi creare un extension method per l'aggiunta di un nostro ipotetico provider che manda l'e-mail.

Esempio 13.19

```
namespace Microsoft.Extensions.Logging
{
  public static class EmailLoggerProviderExtensions
  {
    public static ILoggingBuilder AddEmail(this ILoggingBuilder builder)
    {
      builder.Services.AddSingleton<ILoggerProvider, EmailLoggerProvider>();
      return builder;
    }
  }
}
```

L'uso del namespace dedicato al logging permette in fase di configurazione, come nell'Esempio 13.15, di chiamare `AddEmail` con il supporto dell'IntelliSense e senza aggiungere ulteriori namespace. Un provider non è altro che un factory che genera istanze diverse di `ILogger` per la categoria indicata, come viene mostrato nell'Esempio 13.20.

Esempio 13.20

```
[ProviderAlias("Email")]
public class EmailLoggerProvider : ILoggerProvider
{
  private readonly ConcurrentDictionary<string, EmailLogger> _loggers = new
ConcurrentDictionary<string, EmailLogger>();
  public ILogger CreateLogger(string categoryName)
  {
    // Caching dei logger
    return _loggers.GetOrAdd(categoryName, c => new EmailLogger(c));
  }
  public void Dispose()
  {
    _loggers.Clear();
  }
}
```

Poiché le categorie sono un numero limitato e definito, è buona norma effettuare un caching delle istanze, per non dover istanziare ogni volta

l'oggetto, dato che il logging è uno strumento che viene spesso invocato. L'uso del `ConcurrentDictionary` garantisce la memorizzazione e la creazione di istanze al riparo da problemi di race condition tra i thread. L'uso dell'attributo `ProviderAlias` permette di specificare il nome con il quale fare riferimento all'interno della configurazione, come nell'Esempio 13.17, per indicare che vogliamo un livello minimo di tipo `Error` per questo specifico provider.

Il cuore del nostro provider è l'implementazione di `EmailLogger`, il quale implementa `ILogger` e principalmente il metodo `Log`, che a sua volta ha il compito di memorizzare o, come nel nostro caso, di inviare l'e-mail.

Esempio 13.21

```
public class EmailLogger : ILogger
{
  private string _category;
  public EmailLogger(string category)
  {
    _category = category;
  }
  public void Log<TState>(
    LogLevel logLevel,
    EventId eventId,
    TState state,
    Exception exception,
    Func<TState, Exception, string> formatter)
  {
    // Accedo allo stato come coppia chiave/valore
    var values = state as IEnumerable<KeyValuePair<string, object>>;
    // Serializzo lo stato
    string json = JsonConvert.SerializeObject(values);
    // Ottengo il messaggio
    string message = formatter(state, exception);
    // Invio dell´e-mail...
  }
  public bool IsEnabled(LogLevel logLevel) => true;
  public IDisposable BeginScope<TState>(TState state) => null;
}
```

Possiamo vedere che il metodo `Log` riceve tutte le informazioni: il livello, l'identificativo, lo stato, l'eventuale eccezione e una funzione per ottenere il messaggio formattato. Per una questione di prestazioni,

infatti, spetta a noi decidere se formattare o no il messaggio per rendere più leggera l'operazione di tracciamento. Lo stato può essere di qualsiasi tipo anche se tramite gli extension method dell'Esempio 13.13 è sempre un oggetto che dà una lista di coppie chiave/valore. Nell'Esempio 13.21 non facciamo altro che serializzare in JSON lo stato, per mandarlo ipoteticamente come allegato all'email. L'implementazione di quest'ultima parte è omessa perché esula dagli argomenti trattati in questo capitolo, ma è bene tenere in considerazione che il metodo `Log` potrebbe essere richiamato più volte e dovrebbe quindi essere il più veloce possibile. L'invio dell'e-mail o la memorizzazione su database sono operazioni che sarebbe opportuno fare in asincrono o con un meccanismo di buffer che accumula un po' di messaggi e li invia tutti insieme quando il buffer è pieno oppure ripetendosi a intervalli regolari.

Le potenzialità di un provider sono quindi infinite e, una volta poste le basi, possiamo integrarci con tutti i sistemi, come per esempio Microsoft Azure.

L'integrazione con Azure App Service

Una particolarità che contraddistingue lo sviluppo con le tecnologie Microsoft è sicuramente costituita dalla completezza nella soluzione che mette a disposizione. Oltre al framework web, cioè ASP.NET Core e all'ambiente di sviluppo con Visual Studio, disponiamo anche della possibilità di ospitare i nostri applicativi sulla piattaforma cloud di nome Microsoft Azure. Nello specifico, gli App Service sono un servizio completamente gestito, scalabile e altamente affidabile, dove possiamo distribuire i nostri applicativi web esponendoli con Windows Server e IIS o con Linux e nginx, praticamente con la totalità dei linguaggi e delle tecnologie, tra cui ASP.NET Core. La facilità di distribuzione, la potenza e la presenza anche di piani gratuiti rendono molto appetibile l'ambiente, quantomeno per lo sviluppo.

Quando utilizziamo questo servizio, godiamo anche di una gestione del logging e un aiuto nella diagnostica. Esiste, infatti, la possibilità di scrivere del log su file system oppure su blob, il servizio distribuito di persistenza dei file, fornito sempre da Microsoft Azure. È quindi naturale sfruttare il sistema per usufruire di un salvataggio affidabile e

consultabile senza accesso al server, dato che i servizi PaaS, come quello di App Service, non lo consentono.

L'integrazione tra ASP.NET Core e Microsoft Azure è così forte che nella fase di configurazione, vista nell'Esempio 13.15, troviamo anche un extension method di nome `AddAzureWebAppDiagnostics`. Non solo: chiamarlo è perfino superfluo, perché il provider si installa automaticamente qualora dovessimo eseguire il nostro applicativo sul cloud. L'unica richiesta da fare è quella di pubblicare l'applicativo su un App Service, anche direttamente da Visual Studio, e di abilitare la diagnostica. Questo libro non è una guida sull'utilizzo della piattaforma, ma vogliamo quantomeno mostrare come integrarci nella diagnostica. Non appena entrati nel portale e nella sezione della nostra app, è sufficiente aprire la sezione "Log di diagnostica". Il pannello, visibile nella Figura 13.6, permette di abilitare il logging su file system o su blob.

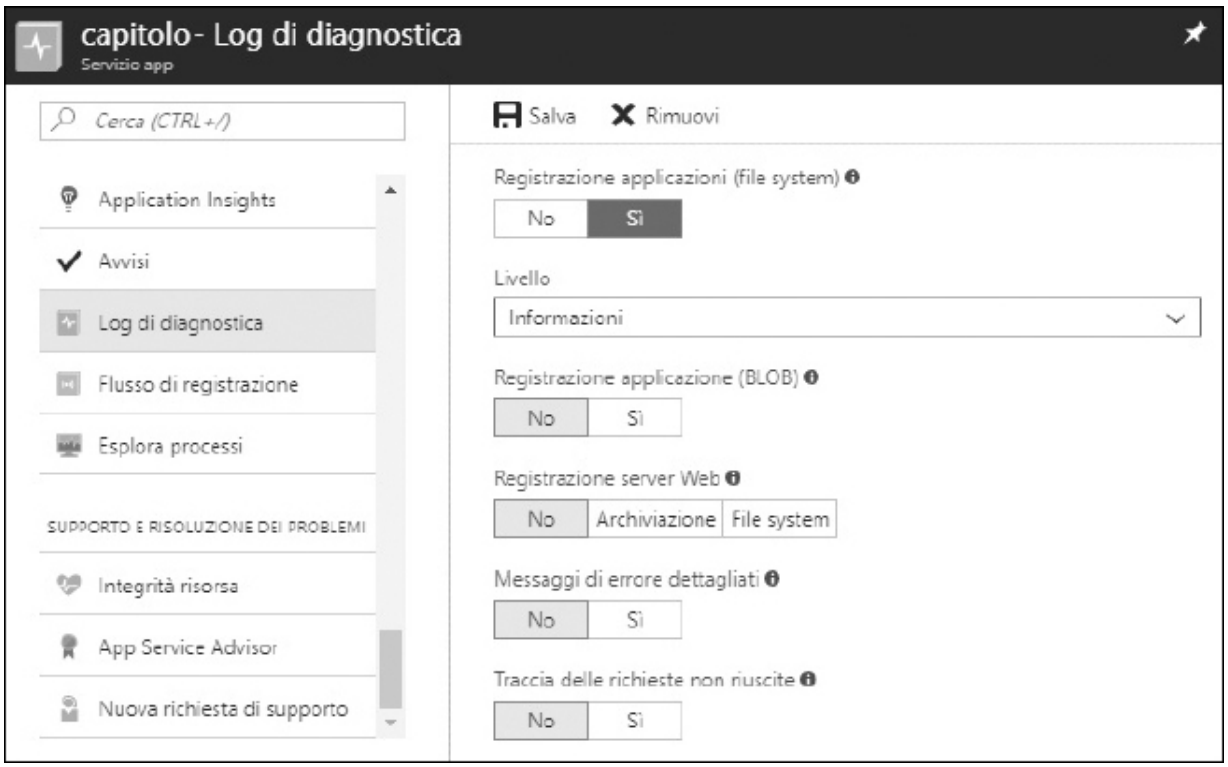


Figura 13.6 – Abilitazione del logging sul portale di Microsoft Azure.

Possiamo cambiare il livello minimo da registrare in qualsiasi momento e, successivamente, possiamo recuperare il log sul blob selezionato o

tramite FTP, navigando nella cartella apposita indicata più sotto nella sezione della Figura 13.6.

Non solo: sempre dal portale possiamo aprire la sezione “Flusso di registrazione”, che permette di attivare una visualizzazione live del logging fornito dal provider. Possiamo vedere nella Figura 13.7 come riusciamo a visualizzare le stesse informazioni ottenute nella Figura 13.5, questa volta attraverso un’interfaccia web.

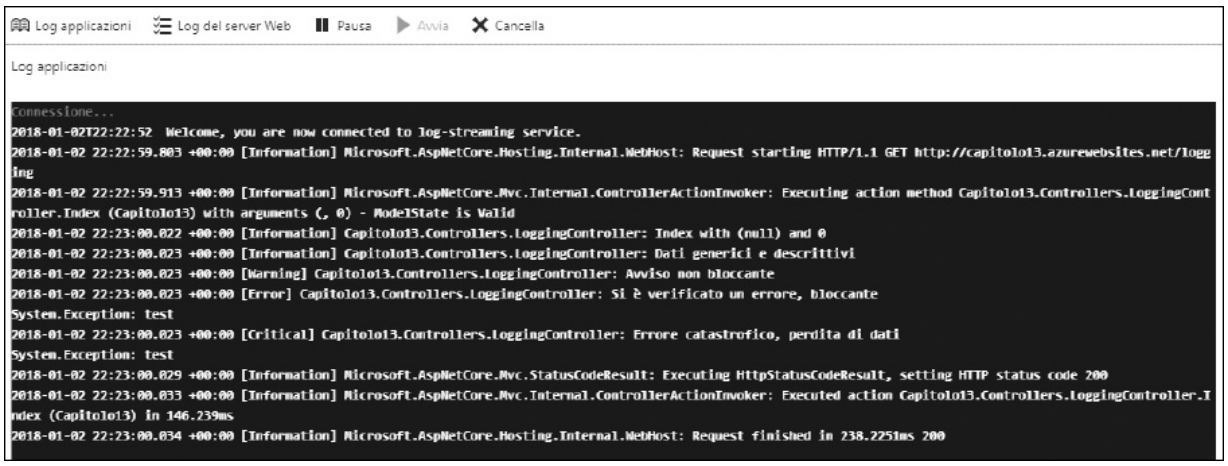


Figura 13.7 – Flusso di visualizzazione live del log sul portale di Microsoft Azure.

L’integrazione, a conti fatti, è talmente banale che diventa pressoché automatico sfruttare questo provider se stiamo distribuendo il nostro applicativo su Microsoft Azure.

Conclusioni

In questo capitolo abbiamo affrontato tutto ciò che ASP.NET Core mette a disposizione affinché il nostro applicativo possa supportare eventuali errori e consenta una facile diagnostica.

Middleware dedicati ci permettono da un lato di avere un supporto durante lo sviluppo, dall’altro di fornire pagine amichevoli per rendere seria l’app. A questo scopo viene in aiuto anche la personalizzazione delle pagine di stato, possibile anche in questo caso tramite view per MVC o per Razor Page. Strumenti dedicati al logging ci permettono di

tracciare tutto ciò che avviene nella nostra app, permettendoci di organizzare le informazioni per livello, categoria, id e stati che possono essere memorizzati o visualizzati in maniera strutturata. Un'architettura a provider ci permette di sfruttare più componenti in grado di gestire contemporaneamente informazioni di tipo diverso e, con poco sforzo, possiamo realizzare un oggetto personalizzato che possa integrarsi con qualsiasi sistema esterno.

Sul tema della personalizzazione vogliamo proseguire anche nel prossimo capitolo, cercando di capire meglio alcuni fra gli aspetti principali che muovono il motore di ASP.NET Core.

14

Estendere ASP.NET Core

Quanto abbiamo affrontato nei capitoli precedenti è sufficiente per cominciare a sviluppare applicazioni con ASP.NET Core, mostrare delle form, offrire API e accedere al database. Ma se vogliamo trarre il massimo dal framework, è necessario approfondire alcune caratteristiche più avanzate che si possono dimostrare fondamentali qualora i nostri progetti crescessero. Diventa sempre più probabile, infatti, poter sfruttare le caratteristiche di dinamicità offerte da ASP.NET Core per la realizzazione di moduli e logiche che possano essere facilmente riutilizzati, nel progetto stesso o in librerie separate, al fine di essere sfruttati più volte.

Nel framework disponiamo di questa possibilità, perciò in questo capitolo partiremo dall'estensibilità offerta dal punto di vista dell'hosting, approfondendo alcuni meccanismi, quindi vedremo come sfruttare la pipeline dei middleware per realizzare componenti, come realizzare librerie contenenti view riutilizzabili, utili anche per una migliore organizzazione dei progetti, fino a entrare nei dettagli di funzionamento di MVC e realizzare filtri personalizzati che lavorino sulla pipeline di esecuzione.

Personalizzare l'host di esecuzione

Nel Capitolo 3 abbiamo affrontato le caratteristiche principali dell'host di un'applicazione ASP.NET Core e come questo venga creato nel `Program.cs` attraverso la chiamata a `WebHost.CreateDefaultBuilder`. L'oggetto restituito offre un'interfaccia alla quale vengono agganciati vari comportamenti predefiniti, che vanno a configurare il motore di dependency injection, a impostare alcuni servizi e ad attivare la classe `Startup`. In quest'ultima, fondamentalmente andiamo a registrare i servizi

e i middleware da utilizzare attraverso i più disparati extension method forniti dalle librerie built-in o di terze parti.

Alcuni dei comportamenti predefiniti, però, non sono inseriti nell'implementazione base di ASP.NET Core, ma sono iniettati in base all'ambiente in cui vengono eseguiti. Abbiamo visto nel Capitolo 13, per esempio, che semplicemente caricando l'applicativo di un ambiente virtualizzato di Microsoft Azure, possiamo utilizzare un provider di logging integrato. Questo è possibile grazie al motore e alla facoltà di iniettare delle dipendenze allo startup dell'applicativo.

Iniettare l'utilizzo di una libreria

Sebbene il pacchetto NuGet `Microsoft.AspNetCore.App` contenga moltissimi assembly, ciò non significa che questi vengano tutti usati, ma solamente che questi sono messi a disposizione per la fase di compilazione. Chiamando poi i rispettivi extension method, come `AddMvc` o `UseMvc`, andiamo a personalizzare l'ambiente host, ma per la registrazione automatica di alcune librerie il motore riserva una via alternativa. Quando avviamo il debug da Visual Studio, per esempio, otteniamo automaticamente la registrazione dell'integrazione con IIS, normalmente non attiva. Questo è possibile grazie a una variabile d'ambiente, di nome `ASPNETCORE_HOSTINGSTARTUPASSEMBLIES` impostata da Visual Studio e contenente una lista di assembly da caricare dinamicamente; Il discovery non è automatico, questo per mantenere alte le prestazioni di avvio del nostro applicativo. Diversamente il motore dovrebbe caricare tutte le dll, per analizzare se sono presenti tipi che soddisfano i requisiti. Questo meccanismo può essere di conseguenza sfruttato anche da una nostra libreria, nel caso in cui non volessimo configurarla manualmente da codice, ma attivarla tramite variabile d'ambiente.

Il primo passo da fare è creare una libreria, referenziare il pacchetto `Microsoft.AspNetCore.Hosting.Abstractions` e implementare con una nostra classe l'interfaccia `IHostingStartup`. L'unico metodo da implementare ci permette di lavorare direttamente con `IWebHostBuilder`, perciò di avere il pieno controllo dell'host. L'istruzione più comune da usare è quella che ci permette di configurare servizi utili alla libreria, come viene mostrato nell'Esempio 14.1.

Esempio 14.1

```
public class MyHostingStartup : IHostingStartup
{
  public void Configure(IWebHostBuilder builder)
  {
    builder.ConfigureServices(s =>
    {
      // Configurazione di un servizio della libreria
      s.AddTransient<MyService>();
    });
  }
}
```

La tecnica è del tutto simile a quanto già facciamo nella classe `Startup`, ma definendo il tutto in una libreria esterna. Occorre successivamente aggiungere, fuori dai namespace, un attributo di assembly, per indicare al motore quale tipo implementa l'interfaccia, come è visibile nell'Esempio 14.2.

Esempio 14.2

```
[assembly: HostingStartup(typeof(MyHostingStartup))]
```

Tramite attributo, anche in questo caso consentiamo una ricerca più rapida, per evitare che il motore si scorra tutte le classi.

Il passaggio successivo richiede la distribuzione della libreria, referenziandola direttamente nel progetto o distribuendola insieme alle altre dipendenze. Non resta altro che valorizzare la variabile d'ambiente prima citata a seconda delle modalità di avvio dell'host. Ai fini di sviluppo, tramite Visual Studio possiamo valorizzare la variabile insieme a ASPNETCORE_ENVIRONMENT come abbiamo già visto nella Figura 3.7. In alternativa, possiamo chiamare la funzione `UseSetting` di `IWebHostBuilder`, come mostrato nell'Esempio 14.3.

Esempio 14.3

```
public static IWebHostBuilder BuildWebHost(string[] args) =>
  WebHost.CreateDefaultBuilder(args)
      .UseStartup<Startup>()
      .UseSetting(WebHostDefaults.HostingStartupAssembliesKey,
        "Capitolo14.Library");
```

L'uso della variabile d'ambiente rimane comunque la tecnica migliore perché ci permette dall'esterno di influenzare i componenti da registrare, facoltà fornita dagli ambienti cloud e da motori di virtualizzazioni come Docker, che verrà affrontato nel Capitolo 22. Con questa tecnica possiamo dunque creare un insieme di servizi in una libreria e registrarli dinamicamente, ma possiamo andare oltre e configurare l'applicazione separatamente.

Configurazione separata dell'app

Nel file di startup, oltre al metodo `ConfigureServices`, dove configuriamo i servizi, disponiamo del metodo `Configure` che, ricevendo un oggetto di tipo `IApplicationBuilder` ci permette di impostare principalmente i middleware da usare, fondamentali per processare le richieste. Con l'Esempio 14.1, purtroppo, non abbiamo questa facoltà, perché ci è concesso solo di configurare i servizi. Viene in aiuto l'interfaccia `IStartupFilter`, che ci permette di eseguire questo ulteriore passaggio. È sufficiente quindi implementare l'interfaccia in una nuova classe e registrarla tra i servizi impostati nell'Esempio 14.1.

L'ambiente di host all'avvio non fa altro che sfogliare tutti i servizi registrati di tipo `IStartupFilter` e li invoca nell'ordine in cui sono stati registrati. Nella loro implementazione, come è visibile nell'Esempio 14.4, dobbiamo restituire un delegato che effettui le operazioni di configurazione di `IApplicationBuilder`.

Esempio 14.4

```
public class MyStartupFilter : IStartupFilter
{
  public Action<IApplicationBuilder> Configure(Action<IApplicationBuilder> next)
  {
    return builder =>
    {
      // Chiamo le altre configurazioni
      next(builder);

      // Aggiungo in coda i miei middleware
      builder.UseMvc();
    };
  }
}
```

La funzione deve restituire un delegato in grado di configurare l'oggetto, ma allo stesso tempo riceve un delegato che invoca le altre implementazioni di `IStartupFilter`, compresa quella presente nello `Startup.cs`. Questo ci permette di intervenire prima o dopo di esse, per dare ordine alla priorità dei middleware.

Tra i servizi utili all'estensione dell'ambiente di host vi è inoltre la possibilità di agganciare servizi in background.

Avviare servizi in background

Middleware, controller e view, tra i principali, sono oggetti che vengono tutti chiamati in causa a fronte di una richiesta da parte dell'utente, perché è questo il compito principale del nostro applicativo. Dato però che esso non è altro che una console sempre attiva, è legittimo pensare che essa possa svolgere altre operazioni, sfruttando thread diversi. Sebbene questo si possa realizzare modificando il `Program.cs`, in .NET Core esiste un approccio standard per realizzare servizi che lavorino in background, indipendentemente dalla parte ASP.NET.

È sufficiente, infatti, implementare l'interfaccia `IHostedService` e registrarla come di consueto nel motore delle dipendenze, nello `Startup` oppure in `IHostingStartup`. Essa dispone di due funzioni `StartAsync` e `StopAsync`, invocate rispettivamente all'avvio, non appena la fase di preparazione dell'applicazione si è conclusa, e all'arresto, non appena viene mandato il segnale di chiusura. Il servizio è quindi ideale per compiere, per esempio, operazioni di setup oppure per avviare un timer che svolga attività di controllo. Nell'Esempio 14.5 possiamo vedere un'implementazione base che prepara proprio un timer, il cui scopo è di controllare un database di utenze e mandare degli avvisi.

Esempio 14.5

```
public class PingCustomerHostedService : IHostedService
{
  private System.Timers.Timer _pingTimer;
  public Task StartAsync(CancellationToken cancellationToken)
  {
    // Ogni 10 minuti
    int interval = 1000 * 60 * 10;
```

```
      _pingTimer = new System.Timers.Timer(interval);
      _pingTimer.Elapsed += OnPingTimer;
      _pingTimer.Start();
      return Task.CompletedTask;
    }
    public Task StopAsync(CancellationToken cancellationToken)
    {
      _pingTimer?.Stop();
      return Task.CompletedTask;
    }
  }
```

L'ambito in cui lavorano le funzioni `StartAsync` e `StopAsync` è disconnesso dal motore web che si mette in ascolto ed è indipendente dalla durata delle attività che eseguiamo all'avvio che, per influenzare il meno possibile altri servizi, dovrebbero eseguire un'attività asincrona breve e, se necessario, lavorare su altri thread per attività lunghe. Questa separazione va tenuta in considerazione nel momento in cui facciamo utilizzo della dependency injection, perché tutto ciò che possiamo ottenere nel costruttore della nostra classe dev'essere stato registrato con ciclo di vita transient o singleton. Ciò che è stato registrato in modalità scope non può essere richiesto, perché non esiste alcuno scope e solitamente questo è circoscritto dal ciclo di vita di una richiesta dell'utente, mancante in questo ambito.

Dobbiamo di conseguenza chiedere nel costruttore un'istanza di `IServiceProvider` e provvedere manualmente alla creazione di uno scope. Tramite quest'ultimo, come è mostrato nell'Esempio 14.6, possiamo poi ottenere un'istanza del servizio che necessita di uno scope, come, per esempio, è un contesto di Entity Framework.

Esempio 14.6

```
public class PingCustomerHostedService : IHostedService, IDisposable
{
  private readonly IServiceProvider _serviceProvider;
  public PingCustomerHostedService(IServiceProvider serviceProvider)
  {
    _serviceProvider = serviceProvider;
  }
  // ... omessi StartAsync e StopAsync
  private void OnPingTimer(object sender, ElapsedEventArgs e)
  {
    // Creazione esplicita dello scope
    using (IServiceScope scope = _serviceProvider.CreateScope())
```

```
      {
        var context = scope.ServiceProvider
          .GetRequiredService<CustomersContext>();

        // Interrogazione clienti
      }
    }

    public void Dispose()
    {
      _pingTimer?.Dispose();
    }
  }
```

Nel codice possiamo vedere la creazione esplicita dello scope, il recupero del servizio interessato e la conseguente distruzione del primo, non più necessario. Nell'esempio possiamo inoltre notare l'implementazione dell'interfaccia `IDisposable`, che possiamo sfruttare per chiudere gli oggetti utilizzati internamente nel momento in cui il container della dependency injection distruggerà il nostro oggetto. Per quanto riguarda la funzione `StopAsync`, va sottolineato che l'operazione asincrona, per garantire la chiusura dell'applicativo, dispone di un tempo massimo di esecuzione di **cinque secondi**, oltre il quale il processo termina comunque. Se proprio è necessario, possiamo aumentare questo limite nella configurazione dell'host, come viene mostrato nell'Esempio 14.7.

Esempio 14.7

```
public static IWebHostBuilder BuildWebHost(string[] args) =>
  WebHost.CreateDefaultBuilder(args)
    .UseStartup<Startup>()
    .UseShutdownTimeout(TimeSpan.FromSeconds(10));
```

Quanto visto ci permette di organizzare il nostro codice in librerie e di agganciarci agevolmente all'ambiente di host, ma questo approccio è utile anche per strutturare controller e view del pattern MVC.

Creare librerie con view Razor

La strutturazione di controller e view all'interno del progetto principale del nostro applicativo è una convenzione che ci permette immediatamente di partire con lo sviluppo delle pagine. I controller sono, di fatto, delle normali classi e possono essere posizionati in altre librerie senza nessuna

particolare configurazione, poiché il motore di ASP.NET MVC cerca in tutti gli assembly referenziati. Questo è particolarmente utile quando i progetti diventano grossi e le aree non sono sufficienti, oppure perché vogliamo poter riutilizzare una parte delle pagine su altri progetti.

Purtroppo, non disponiamo di questa possibilità sulle view, perché esse necessitano di essere compilate e il motore deve conoscere dove sono posizionate e a quale percorso rispondono. Esiste però la possibilità di creare una particolare libreria di tipo **Razor Class Library**, attraverso la seconda finestra di dialogo che si presenta quando scegliamo di creare o aggiungere una nuova ASP.NET Core Web Application, che abbiamo già visto nella Figura 2.6. Non appena creata, la libreria si presenta con un csproj simile al seguente.

Esempio 14.8

```
<Project Sdk="Microsoft.NET.Sdk.Razor">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="2.1.0" />
  </ItemGroup>
</Project>
```

L'uso di un SDK dedicato, già incluso in Visual Studio, evidenziato nell'Esempio 14.8, ci consente di inserire view e di compilarle. Successivamente, la libreria potrà essere referenziata nell'applicativo che ne necessita e, senza effettuare altre operazioni, potremo utilizzare le view Razor contenute al suo interno. La realizzazione di una libreria con controller, model, view e partial view non differisce da quanto facciamo già sull'applicativo principale. Posizioniamo i rispettivi file nelle relative directory rispettando le convenzioni, a seconda dei nomi dei controller, delle action e delle aree. Nella Figura 14.1 possiamo vedere un esempio di strutturazione che rispetta le convenzioni.

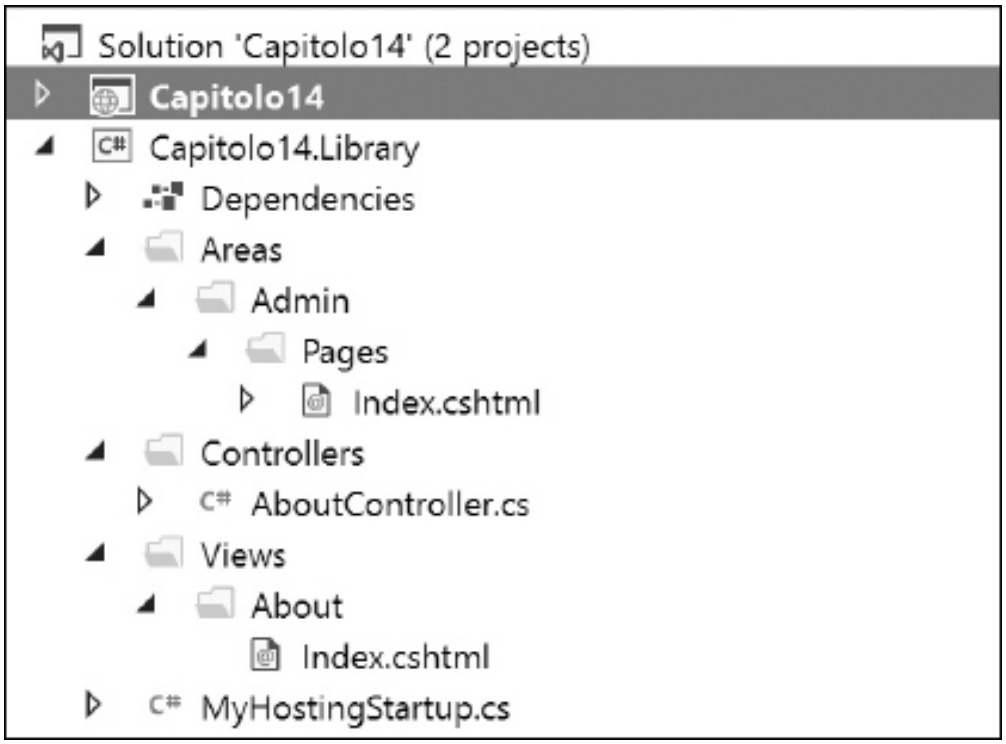


Figura 14.1 – Strutturazione dei file in una Razor Class Library in Visual Studio.

La figura mostra la presenza di un `AboutController` e della relativa view `Index.cshtml`. Dispone inoltre di un'area di nome `Admin` contenente una Razor Page, quindi senza controller, di nome `Index.cshtml`. Le due pagine vengono unite a quanto presente sull'applicativo principale e sono raggiungibili rispettivamente all'indirizzo `/About` (Controller=About, Action=Index, Area=null) e `/Admin` (Page=Index, Area=Admin).

> *Il risultato della compilazione di un Razor Class Library, così come dell'applicativo principale, è sempre composto da due file: nomeLibreria.dll e nomeLibreria.Views.dll. La seconda contiene le view pronte per essere eseguite il più velocemente possibile.*

L'aspetto interessante dell'unione della struttura dei file è il fatto che viene permessa anche la sovrascrittura parziale delle view definite nelle librerie. Ponendo di voler personalizzare la view `About/Index.cshtml`, è sufficiente mettere il medesimo file nello stesso percorso dell'applicativo, lasciando intatta la libreria, e otterremo così la sovrascrittura a runtime di una view,

mantenendo inalterate le altre. Questa tecnica è particolarmente utile in **`One Identity`**, una libreria che affronteremo nel Capitolo 17, la quale porta con sé delle interfacce standard ma che, se necessario, possiamo personalizzare.

Caricare a runtime parti dell'applicazione

Lo sviluppo di librerie è una pratica molto comune e consigliata, perché ci permette di organizzare meglio il codice, di lavorare più facilmente in team e di riutilizzare funzionalità in più applicativi. Non importa se sono delle normali Class Library o delle Razor Class Library, perché in entrambi i casi le dobbiamo referenziare all'interno dell'applicazione principale. Per una questione di ottimizzazione, quando il motore di ASP.NET MVC parte, ispeziona le referenze dell'applicativo stesso e non scorre i file fisicamente presenti nella cartella. Questo meccanismo è gestito dall'`ApplicationPartManager`, un oggetto che raccoglie gli Application Part, i quali, a loro volta, forniscono le feature dell'applicativo, cioè tutti i controller, metadati, tag helper e view disponibili. È grazie a essi che il nostro applicativo MVC è in grado di trovare ed eseguire tutti questi componenti.

L'aspetto interessante è che possiamo intervenire e manipolare quella collezione per decidere di rimuovere o aggiungere **Application Part** per raggiungere i nostri scopi. Poniamo di voler realizzare un applicativo che fornisca delle funzioni base, ma grazie a una cartella di nome `plugin` permetta di inserire le dll che ne arricchiscono le funzionalità dello stesso. Procediamo quindi a creare una Razor Class Library ma, diversamente da quanto è stato fatto nel paragrafo precedente, non la referenziamo nell'applicativo. Per un più facile sviluppo, andiamo nelle proprietà della libreria e, nella sezione `Build Events`, inseriamo il seguente script DOS.

Esempio 14.9

```
set d="$(SolutionDir)bin\$(ConfigurationName)\netcoreapp2.1\plugin"
if not exist %d% mkdir %d%
copy "$(TargetDir)*.dll" %d%
```

Lo script dell'Esempio 14.9 copia tutte le dll generate dalla compilazione della libreria nella directory `plugin` dell'applicativo principale, evitandoci di dover eseguire manualmente questa operazione. Alla luce di quanto abbiamo detto in precedenza, però, questo non è sufficiente, perché ASP.NET MVC non va a caricare questo assembly, ma siamo noi a doverlo fare esplicitamente.

Nello `Startup.cs` riprendiamo il metodo `ConfigureServices` andando a configurare l'`ApplicationPartManager`, dapprima cercando le dll presenti nella cartella, successivamente andandole a caricare, come viene mostrato nell'Esempio 14.10.

Esempio 14.10

```
services.AddMvc()
  .SetCompatibilityVersion(CompatibilityVersion.Version_2_1)
  .ConfigureApplicationPartManager(p =>
  {
    // Ottengo la directory dove sono presenti i plugin
    string dir = Path.GetDirectoryName(typeof(Startup).GetTypeInfo().Assembly.
Location);
    dir = Path.Combine(dir, «plugin»);
    if (!Directory.Exists(dir)) return;
    // Ciclo tutte le dll
    foreach (string file in Directory.GetFiles(dir, ″*.dll″))
    {
      // Carico l´assembly
      Assembly assembly = Assembly.LoadFile(file);
      // Aggiungo l´application part per i componenti
      p.ApplicationParts.Add(new AssemblyPart(assembly));
      // Aggiungo l´application part per le view
      p.ApplicationParts.Add(new CompiledRazorAssemblyPart(assembly));
    }
  });
```

Il codice, commentato nelle sue parti, scorre tutte le dll, chiama il metodo statico `Assembly.LoadFile` per caricare dinamicamente l'assembly e popola la collezione `ApplicationParts`. In essa sono già presenti tutte le Application Part trovate dal motore allo startup e a essa aggiungiamo due tipologie, che vanno entrambe a cercare nell'assembly caricato:

- `AssemblyPart`: cerca i controller e i tag helper contenuti nell'assembly.

- `CompiledRazorAssemblyPart`: cerca le view compilate all'interno dell'assembly.

Poiché nella directory plugin ci potrebbero essere assembly contenenti diverse tipologie di componenti che però ignoriamo, inseriamo entrambe le tipologie. Nel caso di una dll contenente il prodotto della precompilazione delle view, l'uso di `AssemblyPart` sarà superfluo, anche se trascurabile.

Fatta questa modifica, otterremo un applicativo il cui comportamento è dinamico a seconda delle librerie inserite nella directory plugin che vengono caricate in autonomia, senza alcuna distinzione rispetto a quelle referenziate direttamente nel progetto.

I middleware e il contesto HTTP

Nel Capitolo 3 abbiamo introdotto il concetto dei middleware e abbiamo parlato di come siano fondamentali nel processo di gestione di una richiesta. Sono un elemento fondamentale di ASP.NET, perché è solo grazie ai middleware che riusciamo a servire file statici, supportare le funzionalità di routing e di MVC e proteggere le nostre pagine con autenticazione e autorizzazione.

La modularità di ASP.NET è così elevata che un applicativo può anche essere privato di ogni meccanismo e darci comunque il controllo totale di una richiesta HTTP. Nell'Esempio 14.11 possiamo vedere il più semplice degli applicativi che possiamo sviluppare, che otteniamo se in Visual Studio creiamo un progetto ASP.NET Core Web Application di tipo vuoto.

Esempio 14.11

```
public void Configure(IApplicationBuilder app)
{
  app.Run(async (context) =>
  {
    await context.Response.WriteAsync("Hello World!");
  });
}
```

Nell'esempio possiamo notare l'uso della funzione `Run` per configurare un delegato che riceve un'istanza di `HttpContext` (nell'esempio di nome `context`) e tramite quest'ultima scrive in risposta. Ogni operazione che

facciamo sulla risposta avviene in asincrono, perciò sfruttiamo il pattern `async/await` per trarne il massimo dei benefici. L'oggetto `HttpContext`, esposto anche quando siamo in un controller MVC, è il fulcro di ogni richiesta HTTP e contiene i dettagli sulla richiesta, sull'utente, sulla connessione e ci dà accesso alla risposta. Nella Tabella 14.1 sono elencati i principali membri presenti, per fornire un'idea di cosa possiamo fare.

Tabella 14.1 – Principali membri dell'oggetto HttpContext.

Proprietà	Descrizione
`Connection`	Restituisce un oggetto che dà informazioni sulla connessione fisica, sugli IP e i certificati coinvolti
`Request`	Restituisce un oggetto di tipo `HttpRequest` contenente tutte le informazioni della richiesta
`RequestAborted`	Restituisce un `CancellationToken` che permette di supportare la cancellazione delle richieste asincrone
`RequestServices`	Restituisce il riferimento a `IServiceProvider` per poter risolvere le dipendenze esplicitamente
`Response`	Restituisce un oggetto di tipo `HttpResponse` che permette di rispondere all'utente
`Session`	Restituisce un oggetto che permette di memorizzare informazioni di stato
`User`	Restituisce informazioni sull'utente corrente in base al sistema di autenticazione

L'oggetto `HttpRequest` è l'oggetto da guardare se vogliamo accedere ai cookie, alla querystring e alla form, senza l'ausilio di model binding forniti da ASP.NET MVC. Nella Tabella 14.2, proponiamo i membri più importanti, il cui utilizzo è piuttosto intuitivo.

Tabella 14.2 – Principali membri dell'oggetto HttpRequest.

Proprietà	Descrizione
`Body`	Restituisce lo stream di byte della richiesta HTTP ricevuta e da gestire
`Cookies`	Restituisce un dizionario di coppie chiave/valore contenente i cookie r icevuti
`Form`	Restituisce un dizionario che interpreta il `Body` come una form, permettendoci di accedere tramite coppie chiave/valore
`Headers`	Restituisce un dizionario di coppie chiave/valore contenente gli header ricevuti
`Method`	Restituisce il metodo HTTP della richiesta
`Path`	Restituisce il path relativo della richiesta, per esempio `/home`
`Query`	Restituisce un dizionario di coppie chiave/valore contenente i parametri in querystring presenti nell'indirizzo della richiesta

Per quanto riguarda la risposta, i membri essenziali per poter rispondere all'utente sono indicati nella Tabella 14.3.

Tabella 14.3 – Principali membri dell'oggetto HttpResponse.

Membri	Descrizione
`Body`	Restituisce lo stream di byte nel quale possiamo scrivere la risposta
`ContentType`	Permette di scrivere l'header `Content-Type` della risposta
`Cookies`	Restituisce un oggetto che ci permette di aggiungere o cancellare cookie sulla risposta
`Headers`	Restituisce un dizionario di coppie chiave/valore che ci permette di scrivere header in uscita
`StatusCode`	Permette di scrivere lo status code numerico della risposta
`Redirect`	Imposta lo status code a 302 e l'header di `Location` per rimandare l'utente a un altro indirizzo
`RegisterForDispose`	Permette di passare un delegato da invocare quando la risposta è completata

Ai membri esposti da questi principali oggetti troviamo inoltre alcuni extension method che facilitano l'accesso ad alcuni di essi o ne arricchiscono le funzionalità. Per esempio, tramite il namespace `Microsoft.AspNetCore.Http.Extensions` troviamo la funzione `GetEncodedUrl` che restituisce l'URI completo della risposta. Oppure, tramite il namespace `Microsoft.AspNetCore.Http` troviamo la funzione `WriteAsync` utilizzata nell'Esempio 14.11. In questi casi consigliamo di fare affidamento all'IntelliSense di Visual Studio, per poter sfruttare appieno tutte le funzionalità.

Visti gli oggetti fondamentali con i quali dobbiamo lavorare, possiamo ritornare al middleware dell'Esempio 14.1. Grazie alla funzione `Run` indichiamo un middleware completamente personalizzato, ma è l'unico presente nell'applicativo.

Creare un middleware personalizzato

L'oggetto `IApplicationBuilder` sul quale andiamo a configurare i middleware dispone di alcune funzioni per configurare delegati nei quali intervenire nella richiesta. La funzione `Use` ci permette di indicare una funzione asincrona da invocare all'interno della pipeline, espone `HttpContext` ed è l'ideale per scrivere piccoli snippet di codice.

Esempio 14.12

```
app.Use(async (context, next) =>
{
  await context.Response.WriteAsync("Middleware 1 pre");
  // Chiamo il middleware successivo
  await next();
  await context.Response.WriteAsync("Middleware 1 post");
});
app.UseDeveloperExceptionPage();
app.Use(async (context, next) =>
{
  await context.Response.WriteAsync("Middleware 2");
  // Chiamo il middleware successivo
  await next();
});
```

Rispetto a quanto abbiamo visto nell'Esempio 14.11, la funzione `Use` può essere chiamata più volte e nella pipeline di esecuzione otteniamo

l'invocazione dei middleware nell'ordine in cui li abbiamo definiti. Il secondo parametro `next` che le funzioni ricevono dà la possibilità di invocare il middleware successivo, perciò ci dà la facoltà di decidere se lavorare prima o dopo i middleware successivi (che a loro volta possono chiamare i successivi), semplicemente invertendo le istruzioni. Non solo: tramite i costrutti `try/catch/finally` possiamo intervenire inibendo eventuali errori sull'invocazione del next o eseguendo operazioni indipendentemente dall'esito degli altri middleware.

Nell'Esempio 14.11 abbiamo volutamente inserito la chiamata a `UseDeveloperExceptionPage` in mezzo ai nostri due middleware: nel caso di errori la visualizzazione di una pagina, avviene solo se questi sono generati dal secondo e dai successivi middleware. Alla luce di tutte queste considerazioni, richiamando qualsiasi indirizzo del web server, otterremo a video il seguente testo.

Esempio 14.13

```
Middleware 1 pre
Middleware 2
Middleware 1 post
```

È chiaro che rispondere sempre con lo stesso contenuto non trova utilità, soprattutto senza considerare le informazioni della richiesta. L'accesso a `HttpContext` ci permette di vedere la richiesta e quindi la relativa proprietà `Path`, ma le possibilità sono molteplici, e vanno dalla completa scrittura della risposta fino a piccoli comportamenti, come l'aggiunta di un header personalizzato.

Oltre alla funzione `Use` disponiamo di un'altra funzione simile, di nome `Map`, la quale ci permette di distinguere su quale percorso creare una branch che determina una nuova pipeline di middleware di esecuzione.

Nell'Esempio 14.14 possiamo vedere come sfruttare tale istruzioni per richiedere che all'interno di un percorso `/secret` si possa accedere solo in HTTPS, restituendo un 403 (Forbidden) in sua assenza.

Esempio 14.14

```
// Use*** middleware precedenti
```

```
app.Map("/secret", b =>
{
  // Middleware del branch
  b.Use((context, next) =>
  {
    // Verifica HTTPS
    if (!context.Request.IsHttps)
    {
      context.Response.StatusCode = 403;
      return Task.CompletedTask;
    }
    // Chiamo il middleware successivo
    return next();
  });
  b.UseStaticFiles();
});
// Use*** middleware successivi
```

Il percorso richiesto dalla funzione `Map` dev'essere relativo e determina una nuova pipeline. Questo significa che i middleware configurati prima di essa vengono normalmente chiamati, mentre non rientrano quelli successivi. È per questo motivo che all'interno del delegato riceviamo un'istanza di `IApplicationBuilder` (parametro `b`) con il quale costruire un pezzo della pipeline totale. La chiamata a `UseStaticFiles` è quindi fondamentale, anche se questa viene già fatta, ma sul builder principale, successivamente a `Map`. Non commettiamo quindi l'errore di anteporre i middleware che vogliamo siano condivisi, perché dobbiamo sempre considerare il corretto ordine di esecuzione. `UseStaticFiles`, per esempio, non può essere posto prima di `Map`, perché verrebbe eseguito prima della verifica del protocollo. `Map` è quindi da usare solo nelle situazioni in cui vogliamo **differenziare notevolmente la linea dei middleware** da eseguire. Un aspetto interessante, infine, deriva dal fatto che le chiamate a `Map` possono essere innestate l'una all'altra, poiché stiamo sempre lavorando con un'istanza di `IApplicationBuilder`.

Molto simile a `Map` è `MapWhen`, che invece di un percorso vuole un predicato che ci permette in modo libero di valutare i parametri della richiesta. Nell'Esempio 14.15 sfruttiamo questa possibilità per fornire una pagina di stato solo se chi effettua la richiesta è un browser locale e solo per un percorso specifico.

Esempio 14.15

```
app.MapWhen(c =>
```

```
    // Solo IP locale
    System.Net.IPAddress.IsLoopback(c.Connection.RemoteIpAddress)
    // Solo /health
    && c.Request.Path.StartsWithSegments("/health"),
    b =>
  {
    // Unico middleware eseguito
    b.Run(context => context.Response.WriteAsync("Status: OK"));
  });
```

L'uso delle lambda è facoltativo e pertanto potremmo fare uso di funzioni separate, anche se in questo caso risulta più appropriato sviluppare un middleware dedicato.

Creare un middleware come componente

Quando le logiche dei nostri middleware diventano complesse e, soprattutto, quando vogliamo poterle riutilizzare, ha senso sviluppare un componente che ci permetta di adottarle facilmente, allo stesso modo di come sfruttiamo quelli di ASP.NET.

Il primo passo da compiere consiste nel creare una classe che contenga due requisiti: la presenza di un costruttore che accetta un delegato e la presenza di una funzione di nome `Invoke`. Nell'Esempio 14.16 possiamo vedere la struttura base.

Esempio 14.16

```
public class TimerMiddleware
{
  private readonly RequestDelegate _next;
  public TimerMiddleware(RequestDelegate next)
  {
    _next = next;
  }
  public async Task Invoke(HttpContext httpContext)
  {
    await _next();
  }
}
```

Notiamo che gli stessi elementi richiesti dalla funzione `IApplicationBuilder.Use` sono presenti anche qua. Il delegato che ci permette di invocare il middleware successivo viene passato nel

costruttore, mentre il contesto sulla funzione `Invoke`. Questa differenziazione nasce dal fatto che per una questione di ottimizzazione la classe viene istanziata una sola volta, mentre il metodo `Invoke` viene chiamato a ogni richiesta, con un contesto diverso. Ciò significa che dobbiamo **rendere atomica la funzione** e non depositare informazioni di stato sui cambi della classe.

L'assenza di un'interfaccia è dovuta, per permetterci di essere flessibili nella firma del costruttore e della funzione, perché su entrambi godiamo della possibilità di ricevere oggetti tramite la dependency injection. È sufficiente aggiungere le proprie dipendenze a seconda del ciclo di vita che hanno, tipicamente singleton sul costruttore e scope sulla funzione, come è mostrato nell'Esempio 14.17.

Esempio 14.17

```
private IMyService _service;
public TimerMiddleware(RequestDelegate next, IMyService service)
{
  _service = service;
  _next = next;
}
public async Task Invoke(HttpContext httpContext, MyDbContext context)
{
  // ...
```

In alternativa, l'oggetto `HttpContext` ci dà accesso tramite la proprietà `RequestServices` all'`IServiceProvider` permettendoci di ottenere gli oggetti che vogliamo dal motore di dependency injection.

Supponendo di realizzare un middleware che misura i tempi di esecuzione di ogni richiesta, scrivendoli nell'header di uscita, un'ipotetica implementazione potrebbe essere quella mostrata nell'Esempio 14.18.

Esempio 14.18

```
public async Task Invoke(HttpContext httpContext)
{
  // Inizio a cronometrare
  Stopwatch stopwatch = Stopwatch.StartNew();
  try
  {
    await _next(httpContext);
```

```
    }
    finally
    {
      // Fermo il cronometro
      stopwatch.Stop();
      // Controllo che la risposta non sia già stata mandata
      if (!httpContext.Response.HasStarted)
      {
        // Scrivo l´header
        httpContext.Response.Headers[″x-execution-time″] =
          stopwatch.Elapsed.ToString(″g″);
      }
    }
  }
```

L'uso del costrutto `try/finally` ci permette di inserire sempre il nostro header, indipendentemente dall'esito dell'operazione. Risulta fondamentale l'uso della proprietà `HasStarted`, per controllare che la risposta non sia già stata inviata al client, anche parzialmente. Questa situazione si può verificare in presenza di middleware che scrivono file o che forzano la scrittura della risposta senza buffer. Quando questo avviene, scrivere un header genererebbe un errore, situazione che vogliamo quindi evitare.

Una volta realizzato il middleware, non ci resta che utilizzarlo invocando `IApplicationBuilder.UseMiddleware`, il cui scopo è registrare il tipo, ma seguendo lo stile di ASP.NET possiamo realizzare un extension method che renda più comodo l'utilizzo, come quello mostrato nell'Esempio 14.19.

Esempio 14.19

```
namespace Microsoft.AspNetCore.Builder
{
  public static class TimerExtensions
  {
    public static IApplicationBuilder UseTimer(
      this IApplicationBuilder applicationBuilder)
    {
      return applicationBuilder.UseMiddleware<TimerMiddleware>();
    }
  }
}
```

Con questa definizione possiamo chiamare `app.UseTimer` allo stesso modo di come chiamiamo `app.UseMvc`, facendo solo attenzione all'ordine di registrazione. L'uso del namespace relativo al builder facilita ulteriormente

la visibilità dell'extension method e il suo utilizzo. Quello che abbiamo scritto è un middleware semplice, ma vediamo di capire come sfruttarlo in modo più avanzato.

Concetti avanzati sui middleware

I middleware sono un componente fondamentale, non solo dal punto di vista del ruolo che ricoprono ma anche dal punto di vista delle prestazioni. È importante scrivere il componente affinché non ci siano memory leak e non vengano creati oggetti inutili. L'oggetto `HttpContext` dispone della funzione `RegisterForDispose` utile al fine del `Dispose` corretto degli oggetti. Possiamo utilizzarlo come alternativa al costrutto `try/finally`, soprattutto se vogliamo essere certi di effettuare il `Dispose` di un oggetto solo a risposta scritta e non circoscritto alla propria esecuzione, come viene mostrato nell'Esempio 14.20.

Esempio 14.20

```
public async Task Invoke(HttpContext httpContext)
{
  Stream stream = await OpenStreamAsync(httpContext);
  httpContext.Response.RegisterForDispose(stream);
  // ...
```

Una proprietà fondamentale da sfruttare è `RequestAborted`, sempre dell'oggetto `HttpContext`, poiché essa ci dà accesso al `CancellationToken` della richiesta dell'utente. Se la connessione viene interrotta, viene attivato il segnale e, se questo è correttamente gestito, possiamo di conseguenza annullare l'operazione asincrona in corso. È buona norma, infatti, implementare il pattern di cancellazione all'interno di ogni funzione asincrona, ricevendo un `CancellationToken`. Tutti i membri asincroni di .NET accettano questo oggetto per interrompere, se necessario, le comunicazioni socket o operazioni sull'I/O. L'Esempio 14.21 mostra come sfruttare il token per annullare una chiamata HTTP sfruttando la proprietà indicata.

Esempio 14.21

```
public async Task Invoke(HttpContext httpContext,
  IHttpClientFactory clientFactory)
{
  HttpClient client = clientFactory.CreateClient();
  // Passo il token
  await client.GetAsync(″http://www.google.com″,
    httpContext.RequestAborted);

  // Raise dell´eccezione in caso di cancellazione
  httpContext.RequestAborted.ThrowIfCancellationRequested();

  // ...
```

Una volta ottimizzato il codice del middleware, il passo successivo può essere quello di integrarci con la pipeline per permettere un dialogo con gli altri middleware. L'oggetto `HttpContext` dispone a tal fine una proprietà `Features`, una collezione di oggetti popolati in parte dal web server e in parte dai middleware, e rappresenta il punto centrale, contenitore di tutte le informazioni della richiesta che viene soddisfatta, tant'è che oggetti come `HttpRequest` e `HttpResponse` non sono altro che wrapper sulle feature, per facilitarne l'accesso.

Per il nostro esempio definiamo quindi una feature, in genere rappresentata tramite un'interfaccia, dando accesso allo `Stopwatch` della richiesta corrente.

Esempio 14.22

```
public interface ITimerFeature
{
  Stopwatch Stopwatch { get; }
}

public class TimerFeature : ITimerFeature
{
  public Stopwatch Stopwatch { get; }

  public TimerFeature(Stopwatch stopwatch)
  {
    Stopwatch = stopwatch;
  }
}
```

La relativa implementazione non fa altro che esporre il cronometro come da noi richiesto. Nel middleware non ci resta che inserire la nostra feature all'interno del contesto, come viene mostrato nell'Esempio 14.23.

Esempio 14.23

```
public async Task Invoke(HttpContext httpContext)
{
  Stopwatch stopwatch = Stopwatch.StartNew();
  // Aggiungo la feature del timer
  ITimerFeature feature = new TimerFeature(stopwatch);
  httpContext.Features.Set(feature);
}
```

La feature può essere letta da chiunque abbia accesso al contesto e verrà eliminata con esso. Anche in questo caso, un extension method può venire in aiuto per facilitare l'accesso.

Esempio 14.24

```
public static class TimerExtensions
{
  public static Stopwatch GetTimer(this HttpContext httpContext)
  {
    return httpContext.Features.Get<ITimerFeature>()?.Stopwatch;
  }
}
```

Con la presenza di questa funzione, chiunque abbia accesso al contesto, come una action di ASP.NET MVC, può accedere allo `Stopwatch` per effettuare operazioni su di esso.

Viste queste caratteristiche dedicate ad ASP.NET in generale, passiamo a qualcosa di più specifico della parte MVC, per capire come inserire aspetti di Aspect Oriented Programming, tramite i filtri.

Applicare filtri ai controller MVC

MVC è una parte dell'intero framework ASP.NET, che viene azionata attraverso un middleware che porta la richiesta a un livello più avanzato, dove viene mappata su un controller, è identificata una action e i parametri sono convertiti in modelli; il tutto per restituire infine un risultato che molto spesso è una view che genera HTML.

Quando la palla passa a MVC, si aziona una pipeline di invocazione, che dal punto di vista logico è molto vicina ai middleware, ma è più specifica nei confronti dei concetti MVC. Essa è caratterizzata da filtri che vengono

invocati prima e dopo l'esecuzione di una azione e possono essere di natura diversa a seconda delle competenze e del momento in cui vengono invocati e sono:

- **Authorization filter**: i primi tra tutti a essere invocati, hanno il compito di determinare se l'utente corrente è autorizzato.

- **Resource filter**: vengono invocati a seguito dell'autorizzazione, prima dell'azione del controller e dopo di essa, a seguito di tutti gli altri filtri. Vengono avviati prima del model binding e sono l'ideale per implementare meccanismi di cache che inibiscono l'invocazione della action originale.

- **Action filter**: vengono invocati subito prima e dopo la action di un controller e permettono di manipolare gli argomenti della action o il risultato restituito, eventualmente inibendo la chiamata alla action originale.

- **Exception filter**: vengono eseguiti solo quando si verifica un'eccezione non gestita da parte della action o dall'esecuzione del risultato della stessa.

- **Result filter**: sono eseguiti solo quando la action ha avuto successo e ha restituito un risultato. Con i Result filter, possiamo intercettare l'esecuzione prima e dopo il risultato.

L'ordine di esecuzione dei filtri e il momento in cui lavorano è visibile nella Figura 14.2.

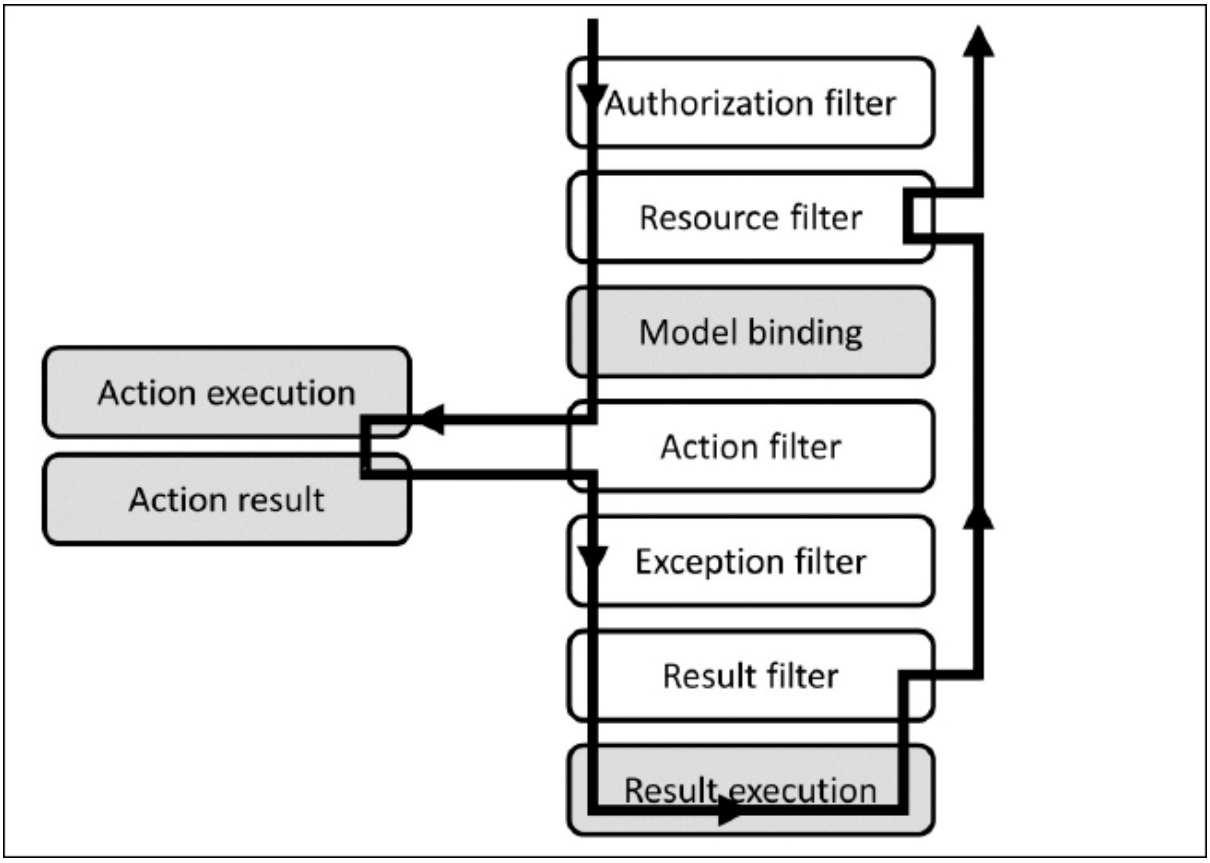


Figura 14.2 – Tipologie di filtri e ordine di esecuzione nella pipeline MVC.

I filtri sono ideali per poter inserire velocemente comportamenti del nostro applicativo e sono la base di molte funzionalità che affronteremo in questo capitolo. Sono oggetti di tipo `IFilterMetadata` nella loro rappresentazione base e dispongono di specializzazioni a seconda della loro tipologia, i cui nomi sono auto-esplicativi: `IAuthorizationFilter`, `IResourceFilter`, `IActionFilter`, `IExceptionFilter` e `IResultFilter`. Per ognuna di esse, esiste una controparte `IAsync***Filter` che espone le medesime funzionalità ma in modo asincrono, restituendo un `Task`.

Per utilizzarli dobbiamo popolare la collezione `Filters` delle opzioni disponibili in fase di configurazione di MVC, come viene mostrato nell'Esempio 14.25.

Esempio 14.25

```
public void ConfigureServices(IServiceCollection services)
{
```

```
    services.AddMvc(o =>
      {
        o.Filters.Add(new RequireHttpsAttribute());
      });
  }
```

Nell'esempio utilizziamo un filtro di tipo autorizzativo, che si accerta che ogni richiesta fatta a MVC avvenga tramite HTTPS e, in caso negativo, effettuerà un redirect. In alternativa al popolamento della collezione `Filters`, possiamo sfruttare gli attributi di .NET, marcando in modo più mirato i controller o le action, a seconda del grado di controllo che vogliamo avere. Il filtro usato nell'Esempio 14.25, oltre a implementare `IAuthorizationFilter` è anche un `Attribute` che possiamo applicare in modo specifico, come viene mostrato nell'Esempio 14.26.

Esempio 14.26

```
  [RequireHttps]
  public class AccountController : Controller
  {
    [ResponseCache(CacheProfileName = "30sec")]
    public IActionResult Index()
    {
      return View();
    }
```

Abbiamo già visto in atto questa tecnica quando abbiamo sfruttato l'attributo `ResponseCacheAttribute` nel Capitolo 8, per effettuare la cache delle response mediante appunto un `IActionFilter` già messo a disposizione.

Nella Figura 14.2 abbiamo visto che ogni tipologia di filtro identifica una fase specifica dove intervenire e determina l'ordine di esecuzione, importante soprattutto se siamo in presenza di più filtri definiti a livello globale, di controller e di action. In caso di più filtri della stessa tipologia, essi vengono eseguiti nell'ordine in cui sono stati inseriti, che nel caso di attributi è a livello globale, di controller e di action. Poiché alcuni di questi filtri sono in grado di lavorare prima e dopo l'esecuzione della action, essi eseguono la post action in ordine inverso una volta che la result è stata eseguita, con la stessa logica che contraddistingue le tipologie filtri, come si evince dalla Figura 14.2. Qualora questa logica non ci dovesse soddisfare, ogni attributo in genere implementa anche l'interfaccia `IFilterOrder` che,

tramite la proprietà `Order`, ci permette di dare una priorità ai figli attraverso un numero con il quale viene poi fatto un ordinamento ascendente, eseguendo per primi i numeri più bassi.

Ora che sappiamo cosa sono e come utilizzare i filtri, non ci resta che vedere come procedere con una implementazione personalizzata.

Creare un filtro personalizzato

Per implementare un filtro personalizzato, teoricamente dovremmo implementare una delle interfacce di specializzazione di `IFilterMetadata`, ma ci vengono in aiuto alcune classi già pronte all'uso. La più comune da usare è la `ActionFilterAttribute`, la quale implementa già per noi un attributo, utile per marcare controller e action, e le interfacce `IActionFilter`, `IAsyncActionFilter`, `IResultFilter`, `IAsyncResultFilter`, `IOrderedFilter`. Contiene tutto quello che serve per intercettare un'azione e il risultato, il tutto in modo sincrono o asincrono. Non dobbiamo far altro che scegliere il momento da intercettare ed eseguire l'override di uno o più membri tra quelli disponibili nella Tabella 14.5.

Tabella 14.4 – Metodi da sovrascrivere per implementare un filtro.

Metodi	Descrizione
`OnActionExecuting`	Permette di intercettare la action di un controller prima che venga eseguita
`OnActionExecuted`	Permette di gestire il risultato di una action del controller
`OnResultExecuting`	Permette di intercettare l'esecuzione del risultato ottenuto dalla action
`OnResultExecuted`	Permette di gestire la risposta prodotta dall'esecuzione di un risultato
`OnActionExecutionAsync`	Permette di intercettare in modo asincrono l'esecuzione di una action, intervenendo prima e dopo
`OnResultExecutionAsync`	Permette di intercettare in modo asincrono l'esecuzione di un risultato, intervenendo prima e dopo

Poniamo quindi il caso di voler realizzare un filtro che inibisce l'accesso a una pagina quando l'utente è anonimo e nelle ore notturne. Per farlo, dobbiamo intervenire prima che l'azione originale venga chiamata e interrompere il proseguimento della pipeline, se è necessario. L'Esempio 14.27 mostra come sovrascrivere `OnActionExecuting`, per poter personalizzare la proprietà `Result` del contesto, al fine di annullare l'esecuzione della action e del suo risultato, fornendone uno personalizzato.

Esempio 14.27

```
public class TimeAttribute : ActionFilterAttribute
{
  public int Hour { get; set; } = 6;
  public override void OnActionExecuting(ActionExecutingContext context)
  {
    bool anonymous = !context.HttpContext.User.Identity.IsAuthenticated;
    bool outOfTime = DateTime.Now.Hour < Hour;
    if (anonymous && outOfTime)
    {
      context.Result = new ViewResult { ViewName = "OutOfTime" };
    }
  }
}
```

Osserviamo come il metodo sovrascritto riceva un contesto che dà accesso all'`HttpContext` visto in precedenza nel capitolo. Possiamo vedere nell'Esempio 14.28 un'implementazione complementare asincrona, che nel nostro caso non porta alcun beneficio poiché sono assenti chiamate asincrone.

Esempio 14.28

```
public override Task OnActionExecutionAsync(
  ActionExecutingContext context,
  ActionExecutionDelegate next)
{
  bool anonymous = !context.HttpContext.User.Identity.IsAuthenticated;
  bool outOfTime = DateTime.Now.Hour < Hour;
  if (anonymous && outOfTime)
  {
    context.Result = new ViewResult { ViewName = "OutOfTime" };
    return Task.CompletedTask;
  }
```

```
    // Chiamo il prossimo filtro o l´esecuzione della action
    return next();
  }
```

Non vi sono differenze degne di nota a seconda del tipo di filtro che possiamo implementare e, nella maggior parte dei casi, abbiamo accesso al contesto e alla manipolazione del risultato. Ciò che può risultare utile, invece, è l'integrazione dei filtri con la dependency injection.

Utilizzare la dependency injection con i filtri

L'uso degli attributi per indicare dove applicare il filtro è molto comodo e intuitivo, ma presenta un difetto: è presente **un'unica istanza dell'attributo** per ogni utilizzo. Questo significa che nella classe non possiamo fare affidamento a membri privati per memorizzare informazioni, magari tra l'executing e l'execution. Oltre a questo non possiamo sfruttare la dependency injection per ottenere servizi utili ai nostri scopi, se non passando dall'oggetto `HttpContext`, il quale espone un'istanza di `IServiceProvider`.

Per ovviare a questo fatto, abbiamo a disposizione due attributi particolari, dei filtri che fanno da tramite e sfruttano la dependency injection. Il primo attributo, di nome `ServiceFilterAttribute`, ci permette di specificare il tipo da istanziare tramite dependency injection.

Esempio 14.29

```
[ServiceFilter(typeof(TimeAttribute))]
public IActionResult Index()
{
  return View();
}
```

Per far sì che tutto funzioni, è necessario che il container conosca il tipo `TimeAttribute`, perciò dobbiamo registrarlo nello `Startup.cs` all'interno del metodo `ConfigureServices`. Grazie alla dependency injection, possiamo arricchire il costruttore del nostro filtro, richiedendo servizi terzi. In questo modo, a ogni richiesta che coinvolge la action dell'Esempio 14.29 viene istanziata la classe `TimeAttribute` con le necessarie dipendenze, per poi essere cestinata. Il difetto di questa soluzione consiste nel fatto che non

possiamo specificare parametri personalizzati, come per esempio dei valori primitivi non presenti nel container.

In alternativa possiamo usare l'attributo `TypeFilterAttribute`, che si differenzia dal fatto che non richiede che il filtro sia registrato e consente l'uso di argomenti personalizzati misti ad argomenti da risolvere tramite la dependency injection. Nell'Esempio 14.30, possiamo vedere un esempio in cui indichiamo di utilizzare il filtro passando il numero delle ore al costruttore.

Esempio 14.30

```
[TypeFilter(typeof(TimeAttribute), Arguments = new object[] { 7 })]
public IActionResult Index()
{
  return View();
}
```

Questo secondo attributo è senz'altro più potente, ma complica un po' l'utilizzo dell'attributo, perciò possiamo valutare un ultimo approccio, che richiede del codice in più, ma più completo e di facile utilizzo.

Possiamo sfruttare un'altra interfaccia di nome `IFilterFactory`, che eredita da `IFilterMetadata`, per implementare un generatore di filtri che può essere sfruttato a livello globale o come attributo. Creiamo quindi un attributo che implementa solo questa interfaccia, come nell'Esempio 14.31.

Esempio 14.31

```
public class Time2Attribute : Attribute, IFilterFactory
{
  public int Hour { get; set; }
  public IFilterMetadata CreateInstance(IServiceProvider serviceProvider)
  {
  //IMyService service = serviceProvider.GetRequiredService<IMyService>();
  return new TimeFilter(Hour);
  }
  public bool IsReusable => false;
}
```

Come per qualsiasi attributo, esponiamo una proprietà di configurazione, in questo caso di nome `Hour`, che possiamo poi sfruttare nella creazione del

filtro tramite `CreateInstance`. Se finora i filtri sono stati al tempo stesso un attributo e svolgevano la duplice funzione di intervenire nella pipeline e di servire alla loro applicazione, con questa tecnica separiamo i ruoli. All'atto della creazione del filtro siamo noi a decidere come creare l'oggetto e, al tempo stesso, disponiamo dell'`IServiceProvider` corrente, che ci permette di risolvere anche dipendenze terze. Non ci resta che implementare `TimeFilter`, limitandoci alla sola implementazione di `IActionFilter` poiché solo a esso siamo interessati.

Esempio 14.32

```
public class TimeFilter : IActionFilter
{
  public int Hour { get; }
  public TimeFilter(int hour)
  {
    Hour = hour;
  }
  public void OnActionExecuting(ActionExecutingContext context)
  {
    bool anonymous = !context.HttpContext.User.Identity.IsAuthenticated;
    bool outOfTime = DateTime.Now.Hour < Hour;
    if (anonymous && outOfTime)
    {
      context.Result = new ViewResult { ViewName = "OutOfTime" };
    }
  }
  public void OnActionExecuted(ActionExecutedContext context)
  {
    // Niente da fare
  }
}
```

Come possiamo vedere, abbiamo solo spostato l'implementazione del filtro dall'attributo verso una classe separata. Questa classe, di conseguenza, può essere utilizzata tramite l'attributo o direttamente istanziata in fase di configurazione dell'applicativo per andare a popolare la collezione `Filters`.

Conclusioni

ASP.NET Core gode di una modularità che ci permette di personalizzare ogni suo aspetto. In questo capitolo abbiamo visto come sfruttare questa caratteristica per sviluppare librerie che s'inseriscono automaticamente

nello startup dell'applicazione o forniscono contenuti come view Razor. Questa caratteristica facilita l'organizzazione dello sviluppo in team e la capacità di riutilizzare i componenti su più progetti. Con la stessa prospettiva, abbiamo visto come realizzare middleware personalizzati che sappiano leggere la richiesta e processare la risposta, in armonia con gli altri configurati. Le feature ci permettono di entrare nelle dinamiche di processamento della richiesta e condividere informazioni con tutti i componenti coinvolti, fino ad arrivare alla view.

Infine, siamo entrati nello specifico della parte MVC e abbiamo analizzato i filtri, e come possiamo configurarli a livello globale o nello specifico di un controller o di un action. Con essi possiamo aggiungere comportamenti in modo flessibile e riutilizzare logiche semplicemente applicando un attributo. Nel prossimo capitolo continueremo questo viaggio di approfondimento sull'estendibilità di ASP.NET Core, esaminando altre caratteristiche del framework MVC.

Startup Configure

```
app.UseMvc(routes =>
{
  routes.MapRoute(
         name: ″default″,
         template: ″{controller=Home}/{action=Index}/{id?}″);
});
```

id

?

```
https://www.miosito.com/
https://www.miosito.com/Home/
https://www.miosito.com/Home/Index/
https://www.miosito.com/Home/Index/1
https://www.miosito.com/Home/Index/abc
```

```
public class HomeController : Controller
{
  public IActionResult Index(string id)
  {
    return View();
  }
}
```

id string

default

1

ToString

Index

/Home/Index

null

id

-
-
-
?

/Home/Index
POST id

Person

```
public class Person
{
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public int Age { get; set; }
}
```

parameter_name.property_name

Person

null

GET

FirstName LastName Age

Address

parameter_name[index]

https://www.miosito.com/home/index?firstName=mario&lastName=rossi

FromHeader

FromQuery

FromRoute

Startup FromForm

```
public ActionResult Index([FromHeader(Name = "Content-Type")] string ct,
  [FromRoute] string id)
{
  return View();
}
```

Content-Type

id

Name

FromBody

FromServices

POST

id

ad hoc

```
public IActionResult Binding([Bind(″FirstName,LastName″)] Person p)
{
  // p.Age sarà sempre 0
  return View();
}
```

Bind

Person

FirstName LastName

Person

```
public class Person
{
  [BindRequired]
  public string FirstName { get; set; }
  [BindRequired]
```

```
    public string LastName { get; set; }
    [BindNever]
    public int Age { get; set; }
  }
```

BindRequired

BindNever

Bind

id

id

FromRoute

```
  public class Person
  {
    [FromRoute]
    public Guid Id { get; set; }
  }
```

BindRequiredAttribute

Creazione di un model binder personalizzato

Person

Person
JsonFormatter

Person

IModelBinder

```
public class PersonBinder : IModelBinder
{
  public async Task BindModelAsync(ModelBindingContext bindingContext)
  {
    if (bindingContext.ModelType != typeof(Person))
      return;

    var body = string.Empty;

    // recupero del body come stringa
    using (var bodyStream = new StreamReader(bindingContext.HttpContext.
Request.Body))
    {
      body = await bodyStream.ReadToEndAsync();
    }

    // body vuoto
    if (string.IsNullOrEmpty(body))
      return;

    // cerco di recuperare il contenuto come json dal jwt ricevuto nel body
    var jsonPerson = JWTHelper.DecodeToken(body);

    if (!string.IsNullOrEmpty(jsonPerson))
    {
      // deserializzo e faccio validazione del modello
      var person = JsonConvert.DeserializeObject<Person>(jsonPerson);

      bindingContext.Result = ModelBindingResult.Success(person);
    }
  }
}
```

IModelBinder
BindModelAsync

Person

POST

Person

string

Result

ModelBindingContext

Person

```
[HttpPost]
public IActionResult Decode([ModelBinder(typeof(PersonBinder))] Person p)
{
  if (p == null)
    return BadRequest();
  return Ok();
}
```

Decode

Person

Decode

Person

ModelBinder

POST

http://aspit.co/bpc

Result

Required

```
// deserializzo e faccio validazione del modello
var person = JsonConvert.DeserializeObject<Person>(jsonPerson);

if (string.IsNullOrEmpty(person.FirstName) ||
    string.IsNullOrEmpty(person.LastName))
{
  // aggiungo l´errore sul ModelState
  bindingContext.ModelState.AddModelError(˝person˝, ˝firstName and lastName
cannot be null or empty˝);
  return;
}
```

ModelBindingContext ModelState

ModelState.IsValid

IModelBinderProvider

```
public class PersonBinderProvider : IModelBinderProvider
{
  public IModelBinder GetBinder(ModelBinderProviderContext context)
  {
    if (context.Metadata.ModelType == typeof(Person))
      return new BinderTypeModelBinder(typeof(PersonBinder));

    return null;
  }
}
```

GetBinder
IModelBinderProvider

PersonBinder

PersonBinderProvider

BinderTypeModelBinder
null

MvcOptions AddMvc
ConfigureServices

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddMvc(options =>
  {
    options.ModelBinderProviders.Insert(0, new PersonBinderProvider());
  });
}
```

PersonBinderProvider

Person

ModelBinderProviders

SimpleTypeModelBinderProvider
ComplexTypeModelBinderProvider
BodyModelBinderProvider

InputFormatter

IEnumerable<T> Dictionary<T,K>

FromServices

Required

Range

```
public interface IPerson
{
  [Required]
  string FirstName { get; set; }
  [Required]
  string LastName { get; set; }
  [Range(18, 100)]
  int Age { get; set; }
}
```

Person

IPerson

ModelMetadataType

```
[ModelMetadataType(typeof(IPerson))]
public class Person : IPerson
{
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public int Age { get; set; }
}
```

ModelMetadataTypeAttribute

Customer

```
[HttpPost]
public IActionResult InterfaceBinding(IPerson person)
{
  return View(model);
}
```

FromServices

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddTransient<IPerson, Person>();
  // registrazione di MVC ed altri servizi...
}
```

```
@model Capitolo15.Models.IPerson
<form asp-action="InterfaceBinding" asp-controller="Home">
  <div class="form-group">
```

```
    <label asp-for="FirstName" class="control-label"></label>
    <input asp-for="FirstName" class="form-control">
    <span asp-validation-for="FirstName" class="text-danger"></span>
  </div>
  <!-- implementazione degli altri campi -->
  <button type="submit">Submit</button>
</form>
```

```
public class InterfaceModelBinder : ComplexTypeModelBinder
{
  public InterfaceModelBinder(IDictionary<ModelMetadata, IModelBinder>
propertyBinder) : base(propertyBinder)
  {
  }
  protected override object CreateModel(ModelBindingContext bindingContext)
  {
    var modelInterface = bindingContext.ModelType;
    var model = bindingContext.HttpContext.RequestServices
                              .GetService(modelInterface);
    return model;
  }
}
```

IModelBinder

ComplexTypeModelBinder

CreateModel ModelType

RequestServices

Person

InterfaceModelBinder

```
public class InterfaceModelBinderProvider : IModelBinderProvider
{
  public IModelBinder GetBinder(ModelBinderProviderContext context)
  {
    if (context == null)
      throw new ArgumentNullException(nameof(context));
    if (context.Metadata.ModelType.GetTypeInfo().IsInterface &&
        !context.Metadata.IsCollectionType &&
        !context.BindingInfo.BindingSource
                  .CanAcceptDataFrom(BindingSource.Services))
    {
      var binders = context.Metadata.Properties
                           .ToDictionary(p => p, p => context.CreateBinder(p));
      return new InterfacesModelBinder(binders);
    }
    return null;
  }
}
```

FromServices

InterfaceModelBinder

Person

SimpleTypeModelBinder

ComplexTypeModelBinder

Address

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddTransient<IPerson, Person>();
  services.AddMvc(options =>
  {
    options.ModelBinderProviders.Insert(0,
                 new InterfaceModelBinderProvider());
  });
}
```

CreateBinder

PersonBinder

IMetadataDetailsProvider

DisplayName

```
public class MetadataDetailsProviderCustom : IDisplayMetadataProvider
{
  public void CreateDisplayMetadata(DisplayMetadataProviderContext context)
  {
    var propertyAttributes = context.Attributes;
    var modelMetadata = context.DisplayMetadata;
    var propertyName = context.Key.Name;

    if (!propertyAttributes.OfType<DisplayAttribute>().Any())
```

```
        modelMetadata.DisplayName = () => char.ToUpper(propertyName[0]) +
                                                propertyName.Substring(1);
    }
  }
```

MetadataDetailsProviderCustom
CreateDisplayMetadata
IDisplayMetadataProvider
Display

DisplayName

IStringLocalizer

IValidationMetadataProvider
CreateValidationMetadata

```
  public class MetadataDetailsProviderCustom : IValidationMetadataProvider
  {
    public void CreateValidationMetadata(ValidationMetadataProviderContext context)
    {
      var propertyAttributes = context.Attributes;
      var modelMetadata = context.ValidationMetadata;
      var propertyName = context.Key.Name;
      if (propertyAttributes.OfType<RangeAttribute>().Any())
      {
        foreach (var attribute in propertyAttributes.OfType<RangeAttribute>())
        {
          attribute.ErrorMessage = $"La proprietà {propertyName} deve rispettare il
          range!";
        }
      }
```

```
    }
  }
```

RangeAttribute

Age

Person

IBindingMetadataProvider

```
  public class MetadataDetailsProviderCustom : IBindingMetadataProvider
  {
    public void CreateBindingMetadata(BindingMetadataProviderContext context)
    {
      if (context.PropertyAttributes != null &&
          context.PropertyAttributes.OfType<RequiredAttribute>().Any())
      {
        context.BindingMetadata.IsBindingRequired = true;
      }
    }
  }
```

BindRequired

Required

MvcOptions

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddMvc(options =>
  {
    options.ModelMetadataDetailsProviders.Insert(0,
        new MetadataDetailsProviderCustom());
  });
}
```

ModelMetadataDetailsProvider

_ViewImports.cshtml

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Microsoft.AspNetCore.Razor.TagHelpers

```
[HtmlTargetElement("if-time", TagStructure = TagStructure.NormalOrSelfClosing)]
public class ConditionalTimeTagHelper : TagHelper
{
  public override void Process(
    TagHelperContext context,
    TagHelperOutput output)
  {
  }
}
```

HtmlTargetElement

<if-time />

<conditional-time />

Process
TagHelperContext

TagHelperOutput

_ViewImports.cshtml

if-time

```
<if-time>
  <h1>Contenuto giornaliero</h1>
</if-time>
```

Process

```
// Evito di scrivere il tag stesso
output.TagName = null;
```

```
int hour = DateTime.Now.Hour;
if (hour > 20)
{
  // Non emetto per intero l´output
  output.SuppressOutput();
}
```

TagHelperOutput

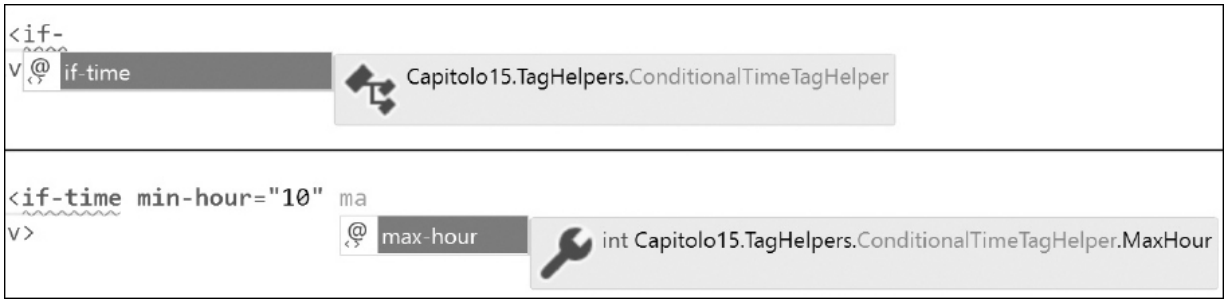
TagName TagMode

SuppressOutput

Esporre proprietà sul tag helper

```
public class ConditionalTimeTagHelper : TagHelper
{
  public int MinHour { get; set; } = 8;
  [HtmlAttributeName(˝max-hour˝)]
  public int MaxHour { get; set; } = 20;
  public override void Process(
    TagHelperContext context,
    TagHelperOutput output)
  {
    int hour = DateTime.Now.Hour;
    if (hour < MinHour ||  hour > MaxHour)
    {
      // ...
```

HtmlAttributeName



@

```
@{
  var interval = new { min = 10, max = 20};
}
<if-time min-hour="@interval.min" max-hour="@interval.max">
  <h1>Contenuto giornaliero</h1>
</if-time>
```

ViewContext

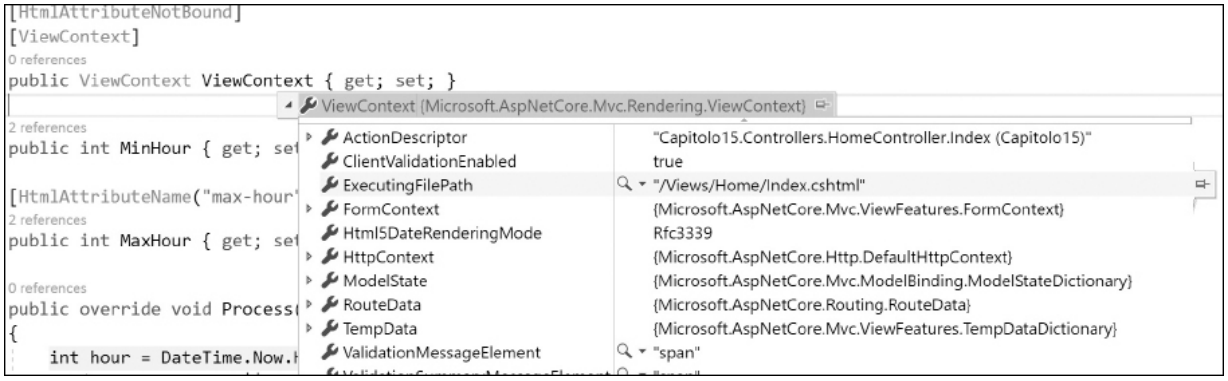
HttpContext ViewData

```
[HtmlAttributeNotBound]
[ViewContext]
public ViewContext ViewContext { get; set; }
```

ViewContext

HtmlAttributeNotBound

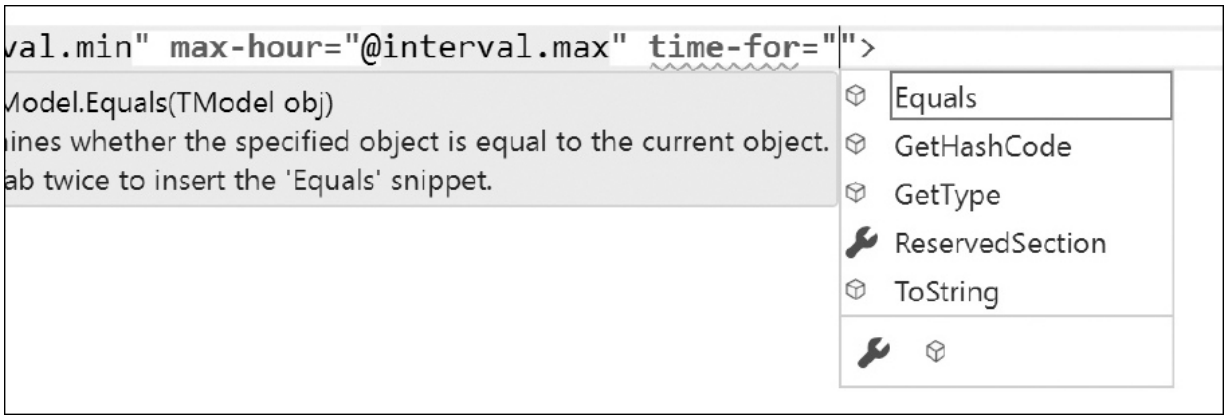
ViewContext



<input />

```
[HtmlAttributeName("time-for")]
public ModelExpression For { get; set; }
```

time-for



- Model
- Metadata
- ModelExplorer

id name

Generare HTML nei tag helper

TagHelperOutput Process

```
// Imposto il tag
output.TagName = ″div″;
// Imposto la chiusura del tag
output.TagMode = TagMode.StartTagAndEndTag;
// Aggiungo un attributo
output.Attributes.Add(″id″, For.Name);

// Sovrascrivo il contenuto
output.Content.SetContent(″Non consentito in questo momento″);

// Aggiungo il contenuto prima e dopo il tag
output.PreElement.SetContent(″pre <div>″);
output.PostElement.SetContent(″post <div>″);

// Aggiungo il contenuto prima e dopo il contenuto stesso
output.PreContent.SetContent(″<div> pre″);
output.PreContent.SetContent(″<div> post″);
```

TagBuilder

```
if (hour < MinHour ||  hour > MaxHour)
{
  // Preparo lo <span>
  var span = new TagBuilder(″span″);
  span.Attributes.Add(″min″, MinHour.ToString());
  span.InnerHtml.Append(″Accesso non consentito″);
  output.Content.SetHtmlContent(span);
}
```

IHtmlContent

TagBuilder

SetHtmlContent

16

Supporto alla globalizzazione e internazionalizzazione

La creazione di un sito web o di un'applicazione in genere che ha necessità di diffondersi a livello mondiale deve poter essere non solo raggiungibile ma anche accessibile a chiunque. Con accessibilità però, non s'intende esclusivamente il supporto per persone che hanno, per esempio, disturbi visivi ma, più in generale, accessibilità significa rendere il servizio che si sta offrendo fruibile con facilità a qualsiasi tipologia di utente finale.

Volendo esporre un caso più concreto, l'inglese è sicuramente la lingua più conosciuta e parlata nel mondo occidentale ma non è detto che sia fruibile da parte di tutti gli utenti, quindi, localizzare i contenuti, per esempio, in italiano, rende il sito stesso accessibile a tutti gli utenti italiani. Un concetto un po' diverso ma strettamente correlato, riguarda invece la visualizzazione di date o di valute, in cui sia il formato sia l'importo possono differire per culture e regioni.

All'interno di questo capitolo, affronteremo come primo tema un'introduzione a quelli che sono i principi e la terminologia che circonda il mondo dell'internazionalizzazione e quindi parleremo di tutti quelli che sono gli strumenti che .NET Core e, in particolare ASP.NET Core, mettono a disposizione per creare applicazioni accessibili. Per riprendere quello che è già stato affrontato nei capitoli precedenti, parleremo anche di come l'internazionalizzazione viene applicata non solo ai contenuti e, tramite Razor, alle view, ma anche di come possiamo localizzare con data annotation e di come possiamo sfruttare i middleware per personalizzarne l'utilizzo.

I requisiti minimi

Per includere il supporto alla localizzazione nelle applicazioni ASP.NET Core, il componente minimo da includere è il pacchetto di NuGet `Microsoft.AspNetCore.Localization`. Ci sono anche altri pacchetti opzionali, per raggiungere un supporto completo alla localizzazione, come:

- `Microsoft.Extensions.Localization`: include la localizzazione, supporto a file di risorse e opzioni per la personalizzazione.
- `Microsoft.AspNetCore.Localization.Routing`: include la localizzazione a livello di routing (l'esempio classico è la specifica della localizzazione in query string come `miosito.com/it-IT/Home/Index`).
- `Microsoft.AspNetCore.Mvc.Localization`: include la localizzazione per tutti i componenti di MVC, compresa la localizzazione delle view.
- `Microsoft.AspNetCore.Mvc.DataAnnotations`: include la localizzazione per tutti gli attributi di validazione e delle data annotation.

Per le applicazioni ASP.NET Core 2.0, non è necessario installare tutti questi pacchetti poiché sono già inclusi all'interno del meta-pacchetto `Microsoft.AspNetCore.All` e pertanto iniziare sarà molto semplice.

Internazionalizzazione, globalizzazione e localizzazione

Quando si parla di internazionalizzazione, in realtà si confondono (o si fa riferimento a loro in modo indifferente) diversi termini: globalizzazione, localizzazione e internazionalizzazione. Questi tre termini sono strettamente correlati ma includono significati diversi poiché il loro utilizzo è differente. L'internazionalizzazione è la parte più semplice,

infatti è l'unione di localizzazione e globalizzazione. La globalizzazione è, in linea di massima, il primo passo che viene eseguito in fase di design dell'applicazione stessa, perché è qui che si impostano le fondamenta che verranno utilizzate poi nei successivi processi di localizzazione. Probabilmente abbiamo già sentito nominare la globalizzazione con un altro termine: "G11n". Questo termine è solamente una abbreviazione in cui il numero "11" è il numero di lettere che separa i caratteri iniziali e finali di "Globalization".

In questo passaggio non scriviamo codice né iniziamo a fare la traduzione dei contenuti. È solo una fase di design, ed è qui che si prendono decisioni riguardanti:

- Le lingue che devono essere supportate.
- La tipologia di supporto: si potrebbe cambiare lingua tramite route specifiche piuttosto che tramite cookie o tramite altri sistemi personalizzati.
- I valori che devono essere utilizzati come placeholder per i valori reali: in questa categoria rientrano i valori che devono essere tradotti in una lingua specifica o per formati diversi di visualizzazione (date piuttosto che valute).
- I formati che devono essere supportati: parlando di valute, possiamo decidere di utilizzare solo l'euro, nonostante l'applicazione possa essere localizzata in inglese e quindi manchi di supporto per la sterlina o per il dollaro.
- La culture di default: in caso un utente provi ad accedere, per vie più o meno supportate e pensate dall'applicazione, a contenuti che non sono disponibili in una lingua o in un determinato formato.

La localizzazione, invece, è un processo che avviene una volta terminata la fase di design e che, per via di com'è costruito ASP.NET Core, ovvero per via della dependency injection, può essere fatta anche una volta che l'applicazione stessa è già nell'ambiente di produzione. Anche in questo caso, la localizzazione ha una sua abbreviazione, "L10n", in cui 10 è

ancora una volta il numero di lettere che separa le iniziali e finali di “Localization”. Questo passaggio, se intendiamo la sola traduzione letterale nelle culture che abbiamo scelto in fase di progettazione, è teoricamente molto semplice e l’unico sforzo necessario è quello di trovare persone adatte a eseguire la traduzione e scrivere un paio di righe di codice. Questo principio di sola traduzione letterale può anche essere trovato abbreviato come “L12y”. Nel caso in cui, invece, si voglia parlare integralmente di localizzazione, bisogna prestare molta attenzione al contenuto che deve essere mostrato: una data in formato americano mostra il mese prima del giorno e, quindi, se cercassimo di applicare una conversione di una data in formato europeo al formato americano, potremmo incorrere in errori a runtime. Proprio per questo la complessità varia in base alle scelte fatte nel processo di globalizzazione e, qualsiasi siano le nostre scelte, sarà opportuno applicare i vari test per verificare che tutto funzioni.

Lavorare con le culture

Per occuparsi di internazionalizzazione, bisogna interagire con le differenti culture. Ogni culture, o locale, in .NET Core, è rappresentata da una serie di regole e formati specifici sia per lingua sia per area geografica e, in generale, lo sviluppatore non deve preoccuparsi di tutte le regole, poiché saranno il framework e il sistema operativo a fornire una implementazione di base.

Ogni culture supportata da .NET Core, viene rappresentata e referenziata tramite quelli che sono chiamati “language tag”, ovvero delle abbreviazioni che permettono di individuare in modo univoco parametri come la lingua e l’area geografica. Questi tag sono ben definiti tramite lo standard RFC 5646 creato e gestito dall’Internet Engineering Task Force (IETF), ovvero lo stesso ente che si occupa di gestire i più grandi standard relativi a internet come OAuth e WebSocket. Un tag è piuttosto semplice e si compone solitamente di tre parti: la prima parte riguarda un identificativo della lingua abbreviato in due lettere in formato ISO 639, la seconda rappresenta l’eventuale dialetto mentre la terza indica, in formato ISO3166-1 sempre abbreviato in due lettere, l’area geografica di riferimento. Per identificare l’italiano, per esempio, si

possono trovare diverse varianti, come “it-CH” che indica la lingua italiana ma con area geografica Svizzera (con i suoi formati), oppure “it-IT” o “it” che sottintendono la lingua italiana parlata in Italia in modo indifferente. Poiché non abbiamo varianti dialettali, per l’italiano non si trovano tag composti da tutte e tre le parti, ma un esempio potrebbe essere fatto con “sr-Latn-BA” che rappresenta la lingua serba parlata in Bosnia-Herzegovina con dialetto latino, piuttosto che “sr-Cyrl-BA” che rappresenta la stessa lingua e la stessa area geografica ma il dialetto di riferimento è questa volta il cirillico.

.NET Core, esattamente come il .NET Framework, include tutti questi concetti in una classe chiamata `CultureInfo`, mentre tutti gli elementi legati ai concetti di internazionalizzazione sono contenuti in classi dell’assembly `System.Globalization`. La classe `CultureInfo` è un po’ cambiata rispetto al .NET Framework classico ma, nonostante non esista più la proprietà `Thread.CurrentThread`, non ne viene alterato il funzionamento, ovvero ogni culture verrà impostata e mantenuta per ogni singolo thread creato e, da .NET Core nello specifico, questo è vero anche per tutto il processo di routing fino a quando la risposta viene consegnata al client. Finora abbiamo parlato solamente di culture, ma analizzando la classe `CultureInfo` troviamo esposte due proprietà differenti:

- `CurrentCulture`: indica come devono essere analizzati date, formati, numeri e valute, ordinamenti e convenzioni per i confronti.

- `CurrentUICulture`: rappresenta ciò che deve essere mostrato nell’interfaccia grafica e riguarda i contenuti come testo e, come vedremo in seguito, HTML.

Poiché queste due culture sono esposte in due proprietà ben separate, non ci sono obblighi di avere gli stessi valori per le stesse proprietà: per esempio, supportare la lingua inglese non implica per forza cose di supportare il formato delle date americano, ma questo dipende molto dalla tipologia di applicazione, dalla logica di business e da quanto è definito nel processo di design. Al contrario, decidere di supportare un differente formato, per esempio di valute, può aggiungere una

determinata complessità all'applicazione stessa: immaginando un sito di e-commerce, i prezzi, che prima dell'aggiunta dell'internazionalizzazione sono stati salvati su un solo campo di un database, non saranno identici sia in dollari sia per euro e sterline, perché non basta cambiare il simbolo, ma bisogna eventualmente applicare una conversione di valuta e capire il formato specifico (il separatore delle migliaia è diverso in base all'area geografica).

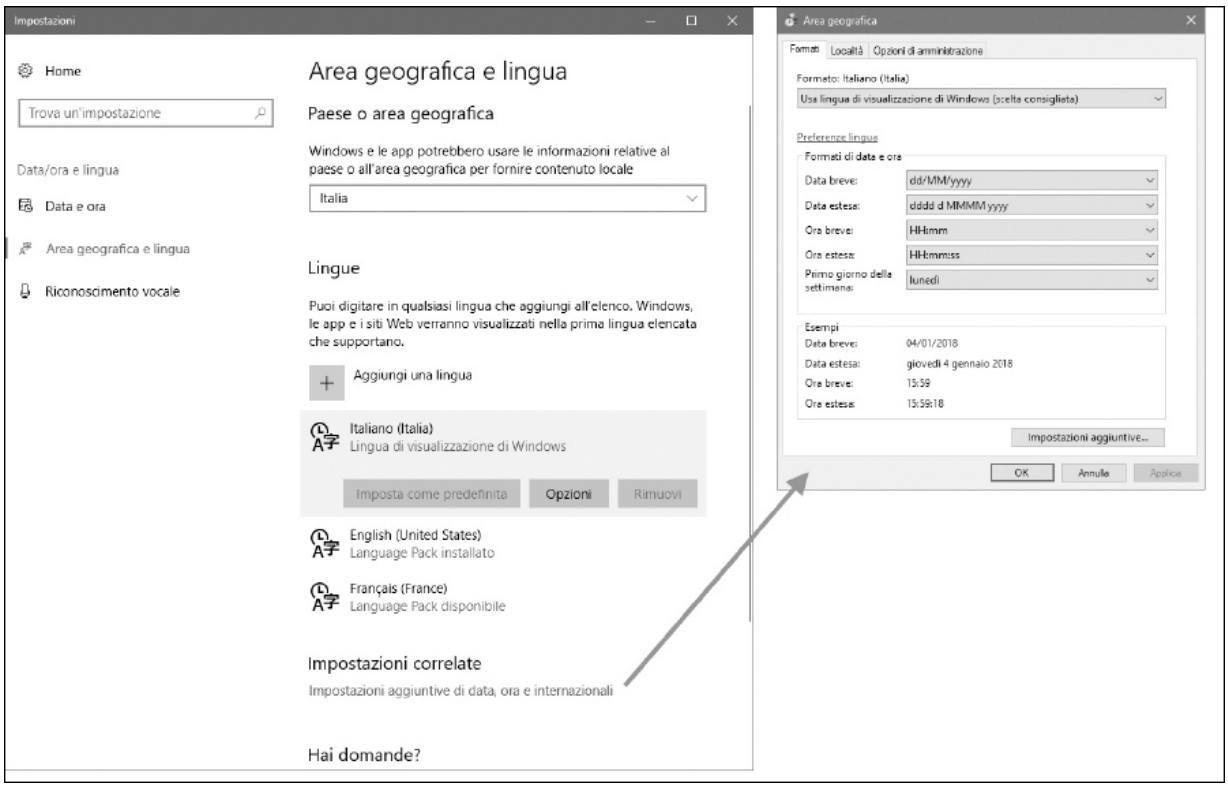


Figura 16.1 – Nelle impostazioni di Windows relative all'area geografica e alla lingua è possibile cambiare i valori corrispondenti alle `CurrentCulture` e `CurrentUICulture` di (ASP).NET Core.

Come viene evidenziato nella Figura 16.1, la `CurrentCulture` e la `CurrentUICulture` sono impostabili direttamente dal sistema operativo tramite gli appositi menù delle impostazioni di "Area geografica e lingua": il primo valore selezionato dall'elenco sarà il valore di default. Da codice, invece, è sufficiente assegnare il language tag all'apposita proprietà, come viene illustrato nell'Esempio 16.1.

Esempio 16.1

```
static void Main(string[] args)
{
  CultureInfo italian = new CultureInfo(″it-IT″);
  CultureInfo.CurrentCulture = italian;
  CultureInfo.CurrentUICulture = italian;

  Console.WriteLine(CultureInfo.CurrentCulture.Name);
  Console.WriteLine(CultureInfo.CurrentCulture.DisplayName);
}
```

L'Esempio 16.1, seppur riferito a un'applicazione console, mostrerà in output il nome della culture scelto, ovvero "it-IT" e il suo nome completo, ovvero "Italiano (Italia)", che è lo stesso valore visibile nelle impostazioni di Windows ed è coerente con quanto appena assegnato, ovvero lingua italiana parlata in Italia.

Per riprendere l'esempio fatto in precedenza riguardo ai numeri e, più nello specifico, alle valute, vediamo come l'Esempio 16.2, nonostante la sua semplicità, possa essere soggetto a errori quasi impensabili.

Esempio 16.2

```
void ChangeCulture()
{
  CultureInfo.CurrentCulture = new CultureInfo(″en-US″);
  NumbersDemo();

  CultureInfo.CurrentCulture = new CultureInfo(″it-IT″);
  NumbersDemo();
}

void NumbersDemo()
{
  string numberAsString = ″10,500″;
  decimal number = decimal.Parse(numberAsString) + 1;
  Console.WriteLine(number.ToString(″C″));
}
```

L'Esempio 16.2 stamperà sulla console due valori, uno per culture, corrispondenti al valore "10,500" incrementato di uno e con il valore della valuta corrispondente alla culture. L'operazione di somma è piuttosto banale, ma i risultati ottenuti sono contrastanti: nel caso in cui la culture selezionata sia quella inglese, l'output generato sarà 10501$, mentre, quando la culture cambia in italiano, l'output diventa 11,50€: il

carattere virgola contenuto nella variabile `numberAsString` assume un valore diverso secondo la culture impostata da separatore delle migliaia a separatore delle unità. È evidente che, quando sviluppiamo applicazioni come quelle bancarie o che hanno a che fare con i numeri in genere, è fondamentale tenere bene a mente che la culture fa la differenza.

Lavorando con le date, i problemi permangono se non prestiamo attenzione alla culture corrente, come evidenziato nell'Esempio 16.3.

Esempio 16.3

```
void DateTimeDemo()
{
  string dateAsString = ″13.01.2018″;
  DateTime date = DateTime.Parse(dateAsString);
  Console.WriteLine(date.ToString(″D″));
}
```

L'Esempio 16.3 effettua il parsing di una stringa contenente una data, in un oggetto di tipo `DateTime` e, in un secondo momento, lo mostra a schermo. Nonostante ci sia già potenzialmente un problema di culture sull'interfaccia grafica per via del fatto che le date potrebbero mostrare giorni e mesi invertiti, passando una culture tipo "en-US" questa funzione fallirà già durante la chiamata a `DateTime.Parse` poiché il numero "13" verrà riconosciuto come mese e, ovviamente, il calendario gregoriano non prevede tredici mesi.

Una possibile soluzione a entrambi i problemi risiede nell'utilizzo di un overload dei metodi `Parse` e `ToString` che accettano un oggetto di tipo `IFormatProvider` come `NumberFormatInfo`, `DateTimeFormatInfo` e `CultureInfo`: specificando la culture, l'elaborazione sarà sempre identica e produrrà un risultato predicibile. Riprendendo l'Esempio 16.2, vediamo una sua possibile soluzione nell'Esempio 16.4.

Esempio 16.4

```
void NumbersDemo()
{
  string numberAsString = ″10,500″;
```

```
    decimal number = decimal.Parse(numberAsString, new CultureInfo(″it-IT″)) + 1;
    Console.WriteLine(number.ToString(″C″, new CultureInfo(″en-US″)));
  }
```

L'ultimo esempio, molto simile ma leggermente diverso per soluzione, riguarda l'uso consistente della culture per quanto riguarda l'ordinamento.

Esempio 16.5

```
void SortingDemo()
{
  string[] states = { ″Washington″, ″Virginia″ };
  Array.Sort(states);

  foreach (string state in states)
  {
    Console.WriteLine(state);
  }
}
```

Considerando come esempio la lingua italiana, l'ordinamento è semplice perché è naturale considerare che la lettera "V" venga prima della lettera "W", ma esistono lingue, come il finlandese, in cui non c'è distinzione tra le due lettere. Pertanto, impostando una culture come "fi-FI", l'output generato risulterà Washington e poi Virginia, perché considerando identica la priorità tra i primi due caratteri, viene valutata la priorità tra i secondi ed evidentemente la lettera "a" viene prima della lettera "i". Stessa cosa si può ritenere valida per altre lingue, come per esempio il bosniaco, in cui le lettere accentate vengono valutate al pari delle stesse lettere senza accento. Anche in questi scenari, la soluzione richiede la sola aggiunta della tipologia di comparativa nel metodo `Sort`, come è illustrato nell'Esempio 16.6.

Esempio 16.6

```
void SortingDemo()
{
  string[] states = { ″Washington″, ″Virginia″ };
  Array.Sort(states, StringComparer.Ordinal);

  foreach (string state in states)
  {
```

```
      Console.WriteLine(state);
    }
  }
```

Tra le possibili soluzioni evidenziate per risolvere i problemi di ordinamento, di numeri e date, si vanno ad aggiungere anche altre due culture di cui non abbiamo parlato: la culture neutrale e la `InvariantCulture`. La culture neutrale è piuttosto intuitiva: riguarda infatti una culture che ha assegnato nel suo tag il solo valore specifico della lingua ma non quello dell'area geografica, come nel caso di "it", che rappresenta in generale l'italiano. L'utilizzo di una culture neutrale è importante quando si vogliono supportare determinate varianti di una lingua ma non tutte, così il sistema è in grado di fare fallback in automatico sulla lingua principale: supponendo di avere l'inglese come lingua, dovremmo prevedere di gestire all'incirca più di quaranta varianti regionali differenti, come quella americana ("en-US"), quella inglese ("en-UK") o australiana ("en-AU") ma, per la maggior parte delle applicazioni, questo non è pensabile. Pertanto, per tutte le altre varianti che non supportiamo, viene comodo gestire in modo generico la lingua inglese ("en"), cosicché, se venisse richiesta, per esempio, la variante "en-ZW", ci sarebbero comunque dei contenuti localizzati.

Ogni culture ha previsto un oggetto padre, il parent, tramite il quale possiamo risalire alla culture più generica: supponendo, per esempio, di avere la variante "it-IT", il suo padre sarà la culture "it", senza la specifica dell'area geografica e questo è vero anche nei casi in cui, come per il bosniaco, venga specificato anche il dialetto. Si può risalire la catena dei parent fino a raggiungere la `InvariantCulture`, una sorta di culture neutrale alla quale però non viene assegnato un vero e proprio tag IETF e la lingua di default è l'inglese. Il padre di una `InvariantCulture` è la `InvariantCulture` stessa.

Esempio 16.7

```
static void Main(string[] args)
{
  CultureInfo.CurrentCulture = new CultureInfo("it-IT");
  while (CultureInfo.CurrentCulture != CultureInfo.InvariantCulture)
  {
```

```
        WriteLine($"CurrentCulture: {CultureInfo.CurrentCulture}");
        WriteLine($"IsNeutral: {CultureInfo.CurrentCulture.IsNeutralCulture}");
        WriteLine($"Parent culture: {CultureInfo.CurrentCulture.Parent}");
        WriteLine("**********");
        CultureInfo.CurrentCulture = CultureInfo.CurrentCulture.Parent;
    }
}
```

Nell'Esempio 16.7 possiamo notare come, indipendentemente dalla culture di partenza si risalirà sempre alla `InvariantCulture` e l'output generato è illustrato nella Figura 16.2.

La `InvariantCulture` può avere senso in quei casi in cui è obbligatorio specificare un parametro indicante la culture ma non si è troppo interessati a gestire l'eventuale valore e, pertanto, tutti gli eventuali metodi, come per esempio il `ToString`, tratteranno l'output nel formato standard inglese. Finora abbiamo parlato di quelle che sono le varie culture e di come .NET Core è in grado di gestirle. Per poter lavorare con le differenti lingue, però, c'è bisogno anche di qualche metodo, come quello offerto dai file di risorse, che consenta il salvataggio delle varie traduzioni.

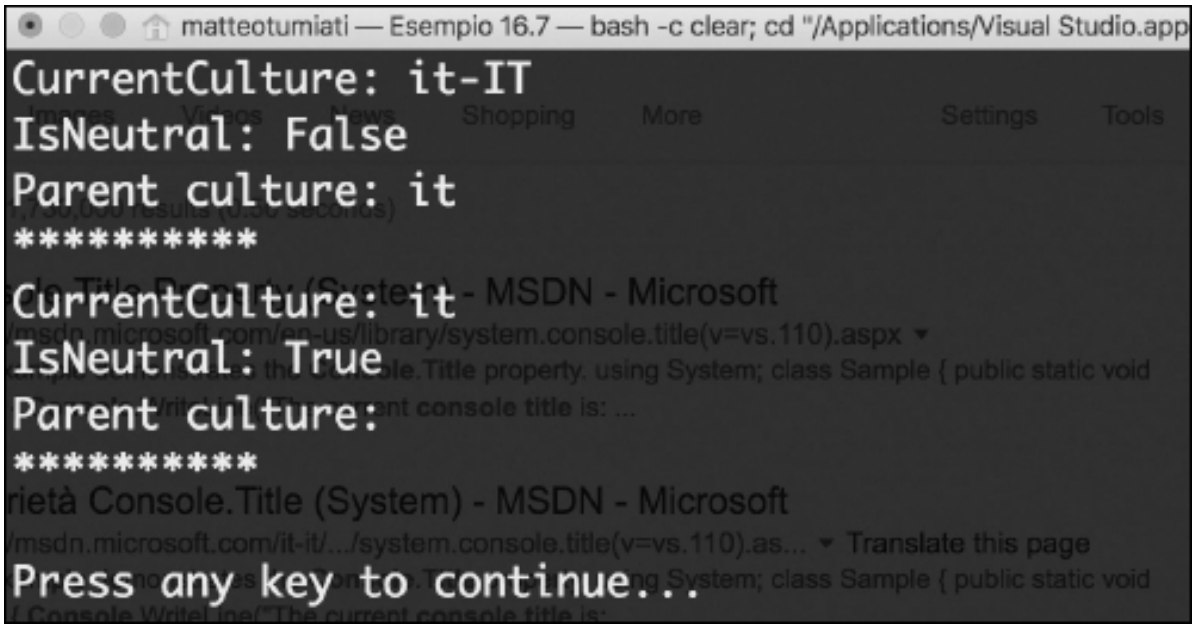


Figura 16.2 – La catena dei padri per il tag "it-IT" risale fino al raggiungimento della `InvariantCulture`.

Utilizzare i file di risorse

Un file di risorse è una sorta di file XML ed è probabilmente il sistema più semplice per salvare i contenuti che l'applicazione può utilizzare in un approccio multi-lingua: mentre link a file, immagini, icone e audio sono utilizzati in un ambiente legato al desktop, per il web ci si concentra principalmente alla sola traduzione del testo. Per creare un nuovo file di risorse, è sufficiente cliccare con il tasto destro sul nome del progetto e aggiungere un nuovo file di tipo "Resource file", come è illustrato nella Figura 16.3.

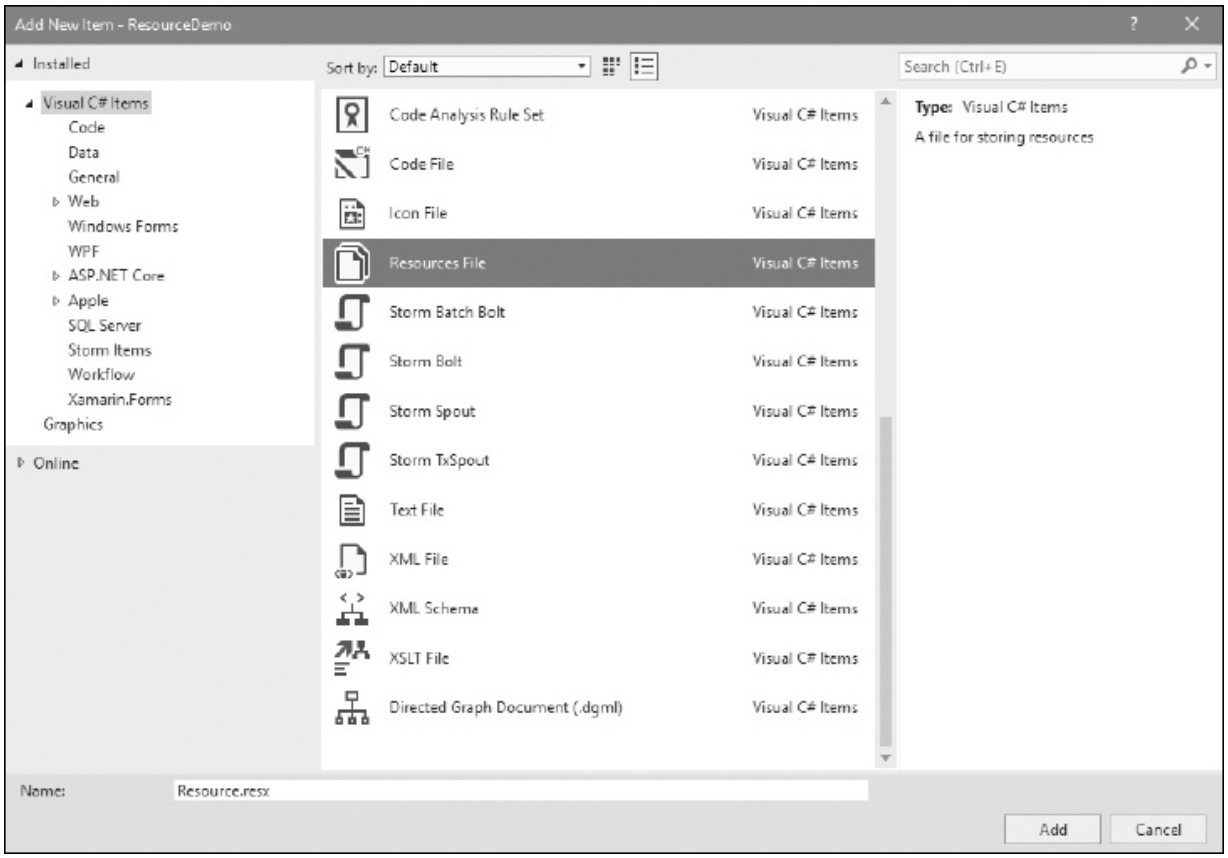


Figura 16.3 – Cliccando con il tasto destro sul progetto è possibile aggiungere un file di risorse.

Il file creato, come abbiamo già anticipato nel paragrafo precedente, è un file XML con una struttura predefinita e, all'interno di Visual Studio, c'è la possibilità di sfruttare il Managed Resources Editor per poterlo

modificare senza dover mettere mano all'XML stesso, come viene mostrato nella Figura 16.4.

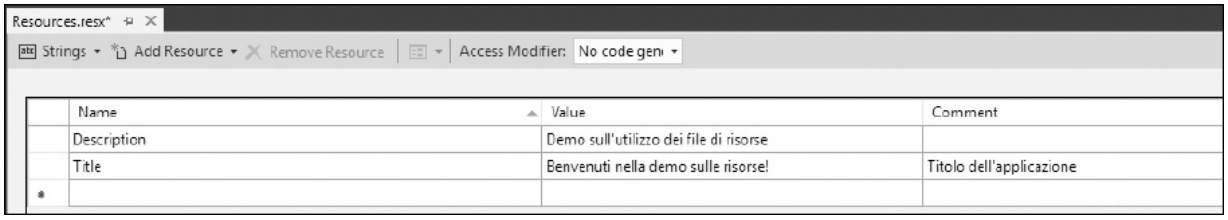


Figura 16.4 – I file di risorse possono essere modificati tramite l'editor visuale di Visual Studio (su Windows).

Il file generato contiene una serie di proprietà come `Name`, ovvero il nome della chiave che verrà utilizzato per referenziare tutte le varie traduzioni, `Value`, che rappresenta la vera e propria traduzione in una determinata lingua e `Comment`, che rappresenta un commento per descrivere meglio quella chiave, ma questo è un valore che può essere omesso. Il file `Resources.resx`, generato da Visual Studio, è composto da uno schema, da un paio di header e, infine, dalle vere e proprie traduzioni, che sono illustrate nell'Esempio 16.8.

Esempio 16.8

```
<data name="Title" xml:space="preserve">
  <value>Benvenuti nella demo sulle risorse!</value>
  <comment>Titolo dell´applicazione</comment>
</data>
```

Saper modificare il valore direttamente tramite l'XML è importante poiché, come abbiamo già annunciato nei primi capitoli, ASP.NET Core e .NET Core in genere sono cross-platform, e pertanto quello che può essere sviluppato su Windows può essere continuato sull'ambiente Visual Studio for Mac di macOS, all'interno del quale però non è presente l'editor visuale e pertanto sarà necessario modificare il file di risorse come se fosse un normale file XML. Tutte le modifiche che vengono fatte sul file di risorse, in modo indipendente dall'utilizzo dell'editor o dalla modalità manuale, vengono riflesse in automatico da

Visual Studio su tutto il progetto perché c'è un watcher che rimane in ascolto di tutti i cambiamenti sul file e, tramite l'utilizzo di un altro file, il `Resources.Designer.cs`, auto-generato e chiamato file di code-behind, possiamo iniziare a referenziare tutte le chiavi, poiché vengono esposte come proprietà pubbliche e statiche all'interno del progetto, come viene mostrato nell'Esempio 16.9.

Esempio 16.9

```
static void Main(string[] args)
{
  Console.WriteLine(Resources.Title);
}
```

Il funzionamento del file di risorse non è cambiato rispetto al passato, dunque uno scenario di migrazione da un sistema esistente è sicuramente più semplice e veloce rispetto a un nuovo utilizzo. La lettura e, come vedremo in seguito, anche la scrittura di un file di risorse, è possibile tramite una classe chiamata `ResourceManager`, che si occupa di astrarre i contenuti del file di risorse e di gestire gli scenari relativi alle culture scelte, consentendo anche di fare il fallback in caso fosse necessario. Nel caso che abbiamo appena affrontato, però, non abbiamo parlato di lingue in modo specifico ma, anzi, abbiamo creato un file di risorse molto generico, come se fosse una sorta di dizionario chiave-valore sempre accessibile. In realtà, questo non è del tutto vero: abbiamo infatti creato un file di risorse che funziona per la `InvariantCulture` perché non è stata specificata alcuna culture. Per specificare che il file di risorse può essere utilizzato con una determinata lingua e area geografica, bisogna applicare una convenzione nel nome, ovvero bisogna creare un file `{nome-a-scelta}.{IETF-tag}.resx`, dove "{nome-a-scelta}" abbiamo visto essere `Resources` nel caso precedente, ma può essere cambiato con qualsiasi altro valore, e `{IETF-tag}` è il valore, opzionale, del language tag da utilizzare. Per creare un file di risorse per l'italiano, per esempio, si può creare un file `Resources.it-IT.resx` e, in base alla culture specificata nella `CurrentUICulture`, il `ResourceManager` decidere se utilizzare un file di risorse specifico per

l'italiano piuttosto che quello per la `InvariantCulture`. Il codebehind per due file di risorse che hanno lo stesso "{nome-a-scelta}" non viene ricreato, poiché si presume che ci sia solo una variazione delle traduzioni e non delle chiavi applicate al file, pertanto rigenerarlo è considerato inutile.

È possibile riscrivere l'Esempio 16.9 facendo uso del `ResourceManager`, come è dimostrato nell'Esempio 16.10, in cui è possibile passare la `CurrentUICulture` come secondo parametro al metodo `GetString`, in modo da referenziare sempre la lingua corretta: i risultati ottenuti saranno identici, ma noteremo in modo più esplicito che cambiando la culture cambierà anche il risultato ottenuto e, qualora non ci sia la chiave ricercata, allora verrà richiamato il fallback.

Esempio 16.10

```
static void Main(string[] args)
{
  ResourceManager rm = new ResourceManager(typeof(Resources));
  var title = rm.GetString("Title", CultureInfo.CurrentUICulture);
}
```

Se andiamo a vedere il contenuto generato dal progetto in fase di build all'interno della cartella `bin`, noteremo che ci sono diverse cartelle con il nome del tag IETF, una per ogni lingua del file di risorse che abbiamo creato, al cui interno ci sarà una dll chiamata `{nome-del-progetto}.resources.dll`. Questa dll è chiamata assembly satellite perché ha solo il contenuto dei file di risorse auto-generato, non contiene codice, non può essere eseguita, può essere elaborata solo durante la compilazione, o a runtime dal `ResourceManager` e può essere iniettata in qualsiasi momento nell'applicazione. Al contrario di quanto abbiamo visto in Visual Studio, in cui ogni file di risorse specificava anche la lingua, l'organizzazione di questi assembly satellite è in cartelle, ma il risultato non cambia perché l'elaborazione è sempre trasparente grazie al `ResourceManager` e, in particolare, ad altre due classi utilizzate dal `ResourceManager` chiamate `ResourceReader` e `ResourceWriter`.

Poiché l'assembly generato dal file di risorse è una normale dll, non è possibile per le classi `ResourceReader` e `ResourceWriter` leggere o

scrivere in modo diretto il file di risorse ma, al contrario, fanno uso di uno stream per caricare eventuali assembly pre-esistenti a runtime. Dobbiamo quindi continuare a utilizzare la `LoadFromAssemblyPath`, che prende in ingresso il percorso assoluto della dll e ritorna l'assembly come oggetto in memoria.

Esempio 16.11

```
void WriteResourceFile(string path)
{
  using (FileStream fs = File.OpenWrite(path))
  {
    using (ResourceWriter rw = new ResourceWriter(fs))
    {
      rw.AddResource("Title", "Titolo della risorsa");
      rw.AddResource("Description", "Descrizione della risorsa");
      rw.Generate();
    }
  }
}
void ReadResourceFile(string path)
{
  using (FileStream fs = File.OpenRead(path))
  {
    using (ResourceReader rr = new ResourceReader(fs))
    {
      IDictionaryEnumerator enumerator = rr.GetEnumerator();
      while (enumerator.MoveNext())
        Console.WriteLine($"{enumerator.Key} : {enumerator.Value}");
    }
  }
}
```

Nell'Esempio 16.11 si è dimostrato come sia possibile scrivere un file di risorse richiamando i metodi `AddResource` e `Generate` oppure leggere il file di risorse sfruttando un classico dizionario chiave-valore, in modo da scorrere tutte le traduzioni inserite. Una volta visto com'è possibile recuperare e lavorare tramite i file di risorse, è ora di iniziare a capire come ASP.NET Core è in grado di recuperare in automatico i contenuti da localizzare tramite l'uso di un'astrazione del `ResourceManager`.

La localizzazione dei contenuti

L'utilizzo dei file di risorse, nonostante sia il metodo più diffuso per quanto riguarda le applicazioni .NET, non è detto che sia il più efficiente o funzionale per quanto concerne le applicazioni moderne e che non devono essere migrate: per le nuove applicazioni, in particolare quelle sulla quale non c'è ancora ben chiara una idea di business e di espansione, può essere dispendioso iniziare a progettare un'applicazione subito in ottica di multi-lingua, pertanto c'è bisogno di metodi alternativi e che siano modificabili in qualsiasi momento. Proprio per questo motivo, e grazie anche al supporto della dependency injection nativa in ASP.NET Core, nascono i localizer, ovvero dei servizi che forniscono la localizzazione a un livello più alto rispetto al `ResourceManager`, che hanno già una implementazione di base nel framework ma sono anche personalizzabili secondo le proprie esigenze e permettono di lavorare non solo con i file di risorse ma anche con database SQL, piuttosto che dati esposti tramite file JSON oppure, oppure proprio per le applicazioni nuove, anche con dati in memoria. L'interfaccia `IStringLocalizer` espone una serie di indexer, proprietà e metodi che possono essere utilizzati dalle classi che la implementano (vedi `StringLocalizer`), in modo tale che si possa fare a meno di referenziare in maniera diretta il `ResourceManager`.

Esempio 16.12

```
LocalizedString this[string name] { get; }
LocalizedString this[string name, params object[] arguments] { get; }
public string Name { get; }
public string Value { get; }
public bool ResourceNotFound { get; }
```

L'Esempio 16.12 mostra alcune delle proprietà esposte dall'interfaccia `IStringLocalizer` che sono piuttosto intuitive: grazie a `Name` possiamo recuperare la chiave, con `Value` il valore della traduzione associata alla chiave, mentre `ResourceNotFound` indica se la risorsa è stata trovata in modo diretto oppure tramite il fallback, così che si possano prendere eventuali decisioni in merito.

Esempio 16.13

```
public interface ICustomService
{
  string SendResponse();
}
public class CustomService : ICustomService
{
  IStringLocalizer _localizer;
  public CustomService(IStringLocalizer<CustomService> localizer)
  {
    this._localizer = localizer;
  }
  public string SendResponse()
  {
    return _localizer["Title"];
  }
}
```

L'esempio 16.13 mostra come fare uso dell'interfaccia `IStringLocalizer` e come possiamo recuperare, tramite uno degli indexer, la traduzione corrispondente alla chiave `Title`. Nel caso questa non venga trovata, verrà restituita la chiave stessa, grazie all'implementazione di base fornita da `StringLocalizer`. Una particolarità che possiamo notare è che all'interno del costruttore non è stata iniettata `IStringLocalizer` ma una versione che fa uso dei generics: questa nuova versione eredita semplicemente dalla versione "normale" ma permette di essere utilizzata in uno scenario a dependency injection, per fare in modo che la stessa interfaccia non venga riutilizzata più volte. Quello che manca per completare è istruire il motore di dependency injection, per dirgli di caricare sia il servizio sia l'implementazione di `IStringLocalizer`, come viene mostrato nell'Esempio 16.14.

Esempio 16.14

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddScoped<ICustomService, CustomService>();
  services.AddLocalization();
}
public void Configure(IApplicationBuilder app,
      IHostingEnvironment env, ILoggerFactory loggerFactory)
{
  app.Run(async (context) =>
```

```
    {
      ICustomService service = context.RequestServices.GetService<ICustomService>
      ();
      await context.Response.WriteAsync(service.SendResponse());
    });
  }
```

Possiamo notare come non ci sia ancora un riferimento esplicito alla parte MVC di ASP.NET Core, e questo è normale perché non siamo obbligati a utilizzarlo. Inoltre, mentre è evidente che il `CustomService` è stato aggiunto alla lista dei servizi, sembra mancare completamente la registrazione per `IStringLocalizer`. La chiamata a `AddLocalization` si occupa di gestire, tra le altre cose, anche della registrazione, pertanto è più che sufficiente per avere una prima risposta. Nel caso in cui volessimo però riutilizzare i file di risorse che abbiamo visto prima, non c'è niente da fare se non prestare attenzione a un paio di dettagli:

- La convenzione dei nomi: ogni file di risorse può essere specifico per una singola classe e deve essere chiamato come `{namespace}.{nome-classe-generica<T>}.{tag-IETF}.resx` se vogliamo identificare la risorsa interamente nel nome, altrimenti ogni pezzo del namespace può diventare una cartella annidata fino ad arrivare all'ultimo livello dove viene specificato il solo tag IETF.

- Il percorso: di default ASP.NET Core si aspetta che le risorse appartengano alla root del progetto.

Poiché di solito i progetti diventano molto complessi con l'avanzare del tempo, per mantenere il progetto pulito e organizzato, possiamo specificare il percorso completo, a partire dalla root, dove saranno presenti i file di risorse tramite l'utilizzo delle `LocalizationOptions`, come viene spiegato nell'Esempio 16.15.

Esempio 16.15

```
services.AddLocalization(options =>
{
  options.ResourcesPath = "Resources";
});
```

Nonostante l'utilizzo sia piuttosto semplice, supponendo che il `CustomService` costruito sia un servizio che deve fare da dispatcher per altri servizi, diventa naturale immaginarsi tanti parametri di tipo `IStringLocalizer<T>` all'interno del costruttore, così da poterne specificare uno per servizio. Ma avere troppi parametri nel costruttore, specie se dello stesso tipo, non è una buona pratica. Per fortuna possiamo approfondire `IStringLocalizer` e vedere com'è fatta dietro le quinte: ogni volta che viene referenziata, il motore di IoC ritorna l'istanza reale, quindi `StringLocalizer`, che però ha nel costruttore un parametro di tipo `IStringLocalizerFactory` che, sfruttando nuovamente IoC, crea una classe di tipo `ResourceManagerStringLocalizerFactory`. Questa nuova classe è in grado di lavorare direttamente con il `ResourceManager` ed è da lì che arrivano, a tutti gli effetti, i contenuti tradotti per ogni chiave richiesta ed è qui che possiamo modificare il comportamento specificando, per esempio, di caricare le risorse da un file JSON piuttosto che da un database SQL. Include inoltre un metodo chiamato `Create`, che permette di creare al volo oggetti di tipo `IStringLocalizer`, secondo quella che è la classe di riferimento, come è dimostrato nell'Esempio 16.16.

Esempio 16.16

```
public interface ICustomService
{
  string SendResponse();
}
public class CustomService : ICustomService
{
  IStringLocalizerFactory _localizerFactory;
  public CustomService(IStringLocalizerFactory localizerFactory)
  {
    this.localizerFactory = localizerFactory;
  }
  public string SendResponse()
  {
    var localizer = localizerFactory.Create(typeof(AnotherService));
    return localizer["Title"];
  }
}
```

Nell'Esempio 16.16 si può notare come a ogni chiamata alla funzione `SendResponse` venga creata una nuova istanza, tramite il metodo `Create`, della factory stessa, per poi ritornare il valore corrispondente alla chiave `Title` inserito nel dizionario.

Nonostante nei paragrafi precedenti abbiamo già introdotto concetti come i file di risorse, utili per salvare le traduzioni nelle varie lingue, e le funzionalità di localizzazione tramite i vari `StringLocalizer`, ci manca ancora di capire come mostrare agli utenti che effettuano le richieste sul nostro sito web i contenuti localizzati nella lingua e culture richiesta.

Il localization middleware

Nei capitoli precedenti abbiamo già parlato nel dettaglio di quelli che sono i middleware e di come funzionano ma, oltre ai middleware più "standard" come quelli relativi a file statici, al routing o all'autenticazione, esiste anche un middleware dedicato alla localizzazione. Ogni chiamata HTTP viene processata da un thread diverso e il middleware della localizzazione lavora in thread isolation. Pertanto, ogni volta che vogliamo processare un determinato URL, siamo certi che verrà restituita la risposta corretta con la culture richiesta dall'utente, anche se le richieste in contemporanea fossero derivanti da più utenti e con più culture. Poiché deve essere garantita la thread isolation, scopriamo anche che nel motore di IoC i servizi `IStringLocalizer` e `IStringLocalizerFactory` sono stati registrati come transient. In questo modo, a ogni request viene generata – e iniettata nelle dipendenze che la richiedono – una nuova istanza.

Nonostante abbiamo già a disposizione dei file di risorse localizzati in più lingue e che abbiamo già visto come cambiare la lingua tramite l'impostazione delle proprietà `CurrentCulture` e `CurrentUICulture`, quello che manca è fare in modo che ASP.NET Core prelevi in automatico la lingua in base a quello che l'utente – o il sistema dell'utente – richiede. Un primo approccio potrebbe essere quello di sfruttare un parametro aggiuntivo nella querystring ma, per questioni di praticità e riusabilità, non è una scelta molto conveniente ed è per questo che entra in gioco il middleware `UseRequestLocalization`, come viene mostrato nell'Esempio 16.17.

Esempio 16.17

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
  app.UseRequestLocalization();
  app.UseMvcWithDefaultRoute();
}
```

Una volta aggiunto il middleware alla pipeline di esecuzione, bisogna configurarne il comportamento. Pertanto, usiamo la classe `RequestLocalizationOptions` per impostare, tramite le sue proprietà pubbliche, quelle che sono le lingue che l'applicazione supporta e la lingua di default (in questo caso specifico, con lingua si intende sia la `CurrentCulture` sia la `CurrentUICulture`). L'Esempio 16.18 lo mette in evidenza.

Esempio 16.18

```
services.Configure<RequestLocalizationOptions>(options =>
{
  options.SupportedUICultures = new List<CultureInfo> {
    new CultureInfo(″it″),
    new CultureInfo(″en ″)
  };
  options.FallBackToParentUICultures = true;
  options.DefaultRequestCulture = new RequestCulture(″es″);
});
```

Nel caso dell'Esempio 16.18, abbiamo impostato come culture supportate l'italiano e l'inglese ma, come culture di default, abbiamo impostato lo spagnolo: nel caso in cui la lingua richiesta sia l'italiano, non ci sono grossi problemi, in quanto è dichiarata come supportata e verrà ricercata all'interno delle risorse disponibili, mentre, qualora facessimo una richiesta con una sua variante, per esempio l'italiano parlato in Svizzera ("it-CH"), il valore ritornato sarà ancora l'italiano, a meno che il fallback sia disabilitato e, in quel caso, verrebbero mostrati contenuti in lingua spagnola, se disponibili, altrimenti il valore previsto dal file di risorse generico per quel file, oppure, in ultima battuta, il valore della chiave di lookup.

Se ispezioniamo la richiesta fatta sul browser, ci accorgiamo che la culture richiesta viene recuperata tramite l'header `Accept-Language` che, in modo completamente automatico, viene valorizzato dal browser con tutti i valori derivanti dal sistema operativo e, con un ordine ben definito, tramite la variabile "q", che ne indica la qualità, viene scelto il valore che ha la preferenza (ovvero un valore di "q" prossimo a uno), come è illustrato nella Figura 16.5.

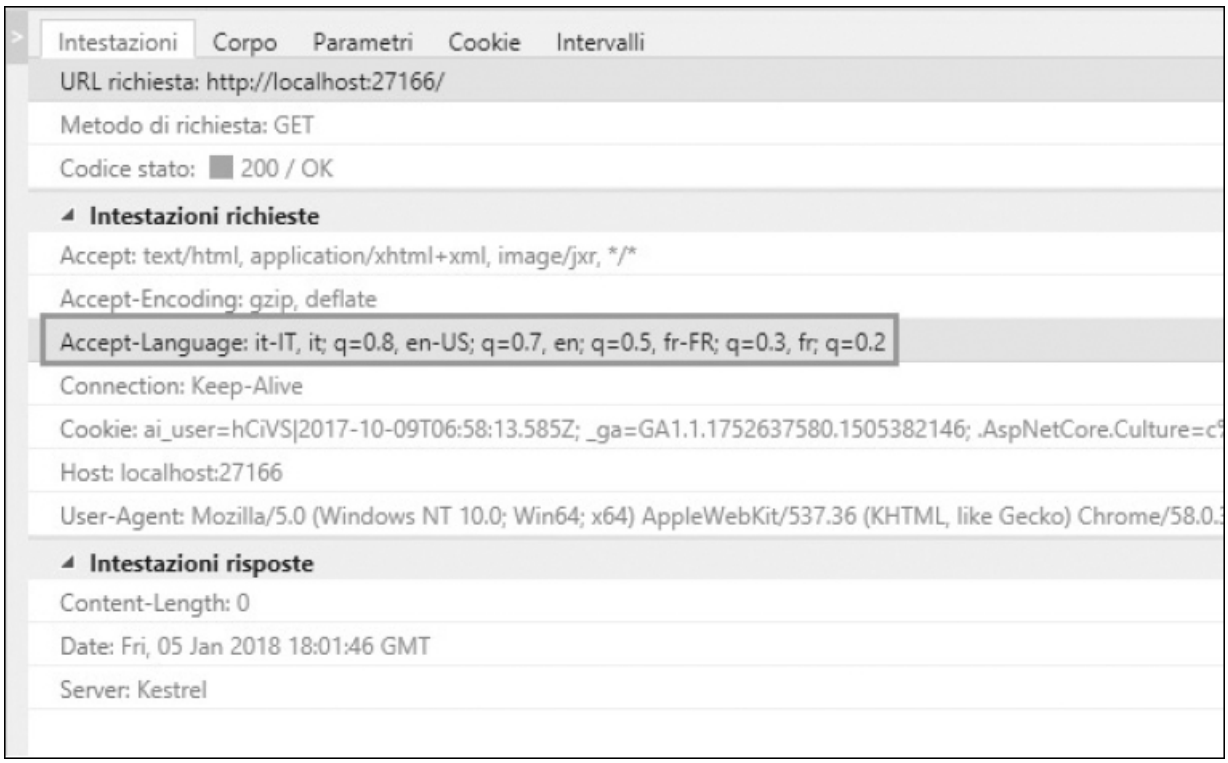


Figura 16.5 – L'header "`Accept-Language`" in una chiamata con il localization middleware abilitato mette in evidenza quali lingue possono essere utilizzare dal framework e secondo quale ordine d'importanza.

Se volessimo dare un po' di precedenza a quanto fatto dal browser, dovremmo intervenire in maniera diretta sulla querystring: passando il parametro `ui-culture`, oppure `culture` in modo indifferente, è possibile fare l'override di qualsiasi valore recuperato dall'header `Accept-Language` e l'intero processo di localizzazione continuerà con il valore della culture recuperato. Qualora il valore della querystring non

esistesse o non fosse valido, verrebbe nuovamente recuperato il valore dall'header.

Entrambi i meccanismi discussi sono validi ma un po' estremi. Pertanto si potrebbe preferire una soluzione intermedia, in cui la culture venga specificata dall'utente e quindi salvata per le request successive: una soluzione basata a cookie potrebbe essere ideale e ASP.NET Core ha previsto un cookie di default, chiamato `.AspNetCore.Culture`, il cui valore è la concatenazione dei tag IETF per culture e culture dell'interfaccia, come è illustrato nella Figura 16.6.

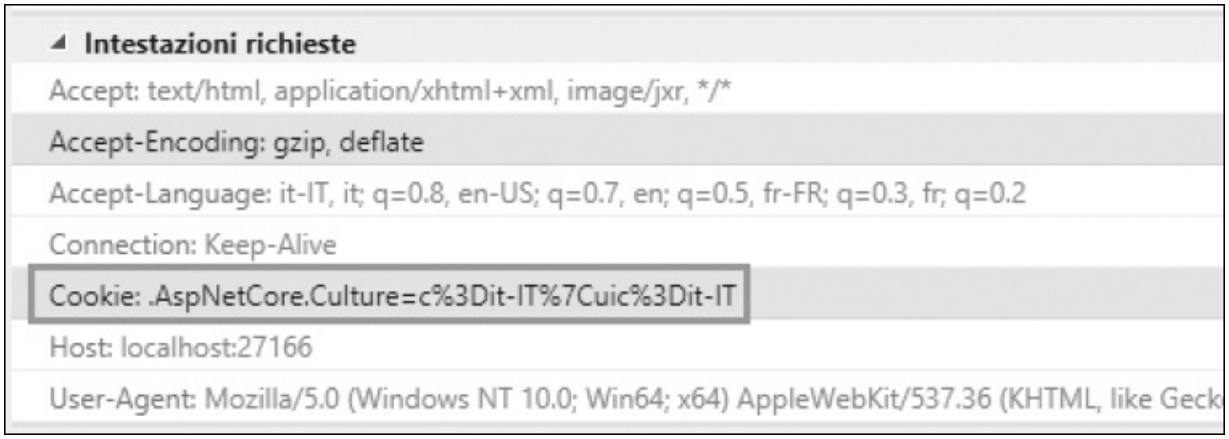


Figura 16.6 – Il cookie di default per la localizzazione di ASP.NET Core è utilizzato per salvare tutti i dettagli relativi alla culture e alla lingua corrente o richiesta durante la chiamata.

Per impostare il valore del cookie, è sufficiente crearlo con il nome predefinito da ASP.NET Core per la localizzazione (oppure uno personalizzato sempre basato sulla localizzazione) e restituirlo nella risposta, come è dimostrato nell'Esempio 16.19.

Esempio 16.19

```
RequestCulture requestCulture = new RequestCulture("it-IT");
var cookie = CookieRequestCultureProvider.MakeCookieValue(requestCulture);
Response.Cookies.Append(CookieRequestCultureProvider.DefaultCookieName, cookie);
```

Le tre pratiche che abbiamo visto sono tutte corrette ma la domanda da porsi ora è: qual è la strategia corretta da utilizzare per le nuove request? La risposta, come sempre nel mondo dell'IT, dipende dalla logica di business, quindi potrebbero essere tutte scelte valide e utilizzabili, così come potrebbero essere tutte sbagliate e potremmo voler preferire un sistema personalizzato. ASP.NET Core è un framework generico, pertanto non può conoscere le logiche di tutte le applicazioni e dà per scontato il fatto che tutte le tecniche che abbiamo visto siano valide e le implementa tramite Request Culture Provider: di default, verrà richiesto il `QueryStringRequestCultureProvider` che controllerà l'esistenza del parametro `ui-culture` o `culture` nella querystring. Qualora non dovesse trovarlo, farà uso del `CookieRequestCultureProvider` che controllerà se per caso esiste il cookie `.AspNet.Culture` e, se dovesse fallire anche quest'ultimo passaggio, proverà a invocare una istanza di `AcceptLanguageHeaderRequestCultureProvider` che farà caricare la culture tramite l'header `Accept-Language`. Per fortuna ASP.NET Core è un sistema formato da tante parti intercambiabili, e pertanto, se volessimo cambiare l'ordine dell'intero processo, sarà sufficiente modificare la proprietà `RequestCultureProviders` all'interno della configurazione delle `RequestConfigurationOptions`, come è dimostrato nell'Esempio 16.20.

Esempio 16.20

```
services.Configure<RequestLocalizationOptions>(options =>
{
  // Setup delle SupportedUICultures...

  // rimozione di tutti i provider di default
  options.RequestCultureProviders.Clear();

  // aggiunta dei provider nell´ordine personalizzato
  options.RequestCultureProviders.Insert(0, new
    AcceptLanguageHeaderRequestCultureProvider());
  options.RequestCultureProviders.Insert(1, new
    CookieRequestCultureProvider());
});
```

Nel caso in cui si vogliano creare URL con la localizzazione in modo che siano SEO-friendly, l'aggiunta del parametro `culture` nella querystring

non è l'ideale ma, anzi, sarebbe meglio avere la culture già all'interno della route scelta. Come viene dimostrato nell'Esempio 16.21, per realizzare un sistema di questo tipo è sufficiente aggiungere il provider dedicato alla gestione delle route in prima posizione, così da dargli la giusta priorità, quindi bisogna creare una route dedicata a livello di MVC.

Esempio 16.21

```
public void ConfigureServices(IServiceCollection services)
{
// Aggiunta della localizzazione...
  services.Configure<RequestLocalizationOptions>(options =>
  {
    // Aggiunta delle SupportedUICultures...
    // Inserimento del provider dedicato
    options.RequestCultureProviders.Insert(0,
      new RouteDataRequestCultureProvider());
  });
  services.AddMvcCore();
}
public void Configure(IApplicationBuilder app)
{
  app.UseRequestLocalization();
  app.UseMvc(routes =>
  {
    routes.MapRoute(
      name: "defaultWithCulture",
      template: "{ui-culture}/{controller=Home}/{action=Index}/{id?}"
    );
  });
  app.UseMvcWithDefaultRoute();
}
```

Proseguendo su questa linea di principio, non è complesso realizzare un provider personalizzato: è solamente una classe che implementa l'interfaccia `IRequestCultureProvider`, ma poiché dipende dalla logica della propria applicazione, lasciamo al lettore l'implementazione concreta e la relativa documentazione ufficiale, disponibile all'indirizzo: http://aspit.co/bns.

Localizzare i contenuti e poterli gestire tramite i vari provider potrebbe già essere sufficiente per la maggior parte delle applicazioni;

però esistono diversi scenari, come vedremo a breve, in cui le stesse view possono aver la necessità di mostrare componenti differenti.

La localizzazione delle view

Quello che abbiamo trattato finora si occupa in genere di localizzare i contenuti che possono essere serviti da un controller o, più in generale, da un servizio, ma quando si tratta di localizzare porzioni di view o, addirittura, codice HTML, bisogna introdurre un nuovo concetto: gli HTML localizer sono servizi che scrivono in modo diretto all'interno delle view e che non fanno uso dell'encoding in HTML di Razor. Il loro funzionamento, nonostante l'interfaccia `IHtmlLocalizer` non la erediti in modo esplicito, è identico a quanto abbiamo visto già lavorando con `IStringLocalizer`, perché la creazione di un `IHtmlLocalizer` viene fatta tramite una istanza di `IHtmlLocalizerFactory` che, nell'implementazione predefinita, crea a tutti gli effetti uno oggetto di tipo `IStringLocalizer`. Poiché, però, viene richiesto in modo esplicito un qualcosa che possa lavorare con l'HTML, l'istanza creata viene wrappata all'interno di un oggetto che abbia una minima conoscenza dei meccanismi di rendering sulle view e che sia in grado di recuperare il codice HTML salvato nei file di risorse. Oltre all'uso, anche la registrazione è molto simile, ma richiede una variazione durante la fase di startup, come è mostrato nell'Esempio 16.22.

Esempio 16.22

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddMvc().AddViewLocalization();
}
```

In questo caso si va a lavorare in modo esplicito con le view, pertanto la dipendenza da MVC è obbligatoria. All'interno della vista che ha i contenuti localizzati, basta iniettare la dipendenza su `IHtmlLocalizer`, specificando come tipo generico il controller di riferimento, come viene mostrato nell'Esempio 16.23. La risoluzione viene fatta tramite IoC in

modo corretto poiché tutte le classi sono già state registrate con la chiamata all'extension method `AddViewLocalization` dell'esempio precedente.

Esempio 16.23

```
@model string
@inject IHtmlLocalizer<HomeController> localizer

<h1>@localizer[˝Title˝]</h1>
<p>@localizer[˝Description˝, Model]</p>
```

Così come per la `ILocalizedString`, l'interfaccia `ILocalizedHtmlString` espone un indexer che accetta un secondo parametro opzionale a cui passare eventuali dati che vengono sostituiti ai placeholder contenuti nel valore associato alla chiave. Al contrario del valore stesso, però, i parametri opzionali non subiscono l'encoding. Per dimostrarlo, supponiamo di avere a disposizione una chiave `Title` il cui valore è "Benvenuto {0}": al posto del placeholder "{0}" verrà aggiunto, grazie alla view, il nome della persona che sta effettuando il login attraverso una form. Per capire se l'applicazione è vulnerabile a possibili attacchi di scripting, il primo test è provare a inserire al posto del nome la classica chiamata al metodo `Alert` di JavaScript, così, qualora dopo la `POST` della form si dovesse vedere il popup comparire, sapremmo che abbiamo dei problemi da risolvere. In realtà, come si può notare nella Figura 16.7, il contenuto non è stato elaborato ma solamente renderizzato come un normale campo di testo.

Figura 16.7 – Il rendering dei parametri associati al valore di una chiave di lookup con codice JavaScript non produce alcuna dialog poiché viene

mostrato come solo testo.

Se invece cambiassimo il valore della chiave `Title` da “Benvenuto {0}” allo stesso script JavaScript appena passato come parametro, noteremo tutto un altro risultato.

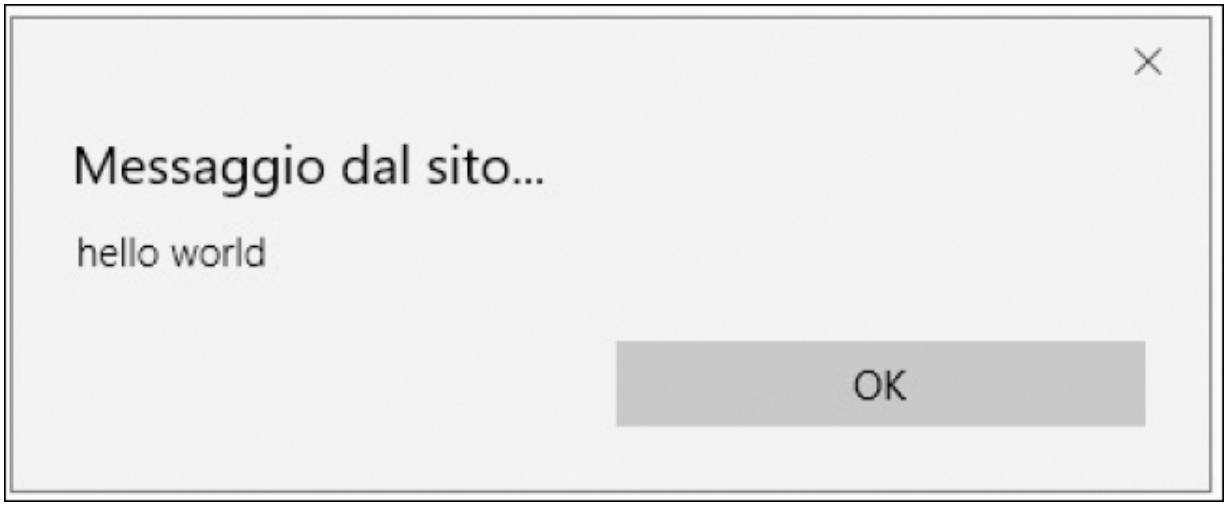


Figura 16.8 – Il rendering del valore di una chiave di lookup con codice JavaScript, invece, produce una dialog.

Questa differenza è dovuta al fatto che il render sta avvenendo molto presto nella costruzione della view, quindi tutto il JavaScript viene elaborato fin da subito, causando la comparsa del messaggio a schermo. Nonostante in questo caso non ci siano grossi problemi perché abbiamo scritto personalmente il file di risorse e quindi siamo noi stessi responsabili di eventuali problematiche a esso associate, a meno di applicazioni particolari, come i CMS, è fortemente consigliata la localizzazione di soli contenuti e non di codice HTML nelle risorse.

Un approccio molto simile a quello che abbiamo appena affrontato con gli HTML localizer lo si può avere con i view localizer, ovvero dei servizi che lavorano in modo specifico con Razor. L’interfaccia che verrà iniettata in questo caso è `IViewLocalizer`, che deriva in modo diretto da `IHtmlLocalizer` e non definisce nuove funzionalità e, pertanto, quanto visto finora è ancora valido. In più, però, eredita anche un’altra interfaccia, ovvero `IViewContextAware`, che ha a disposizione il metodo

`Contextualize`, che viene utilizzato ogni volta che del contenuto deve essere mostrato nella view, così che, qualora ci fosse una dipendenza da qualche altro servizio durante l'utilizzo, il metodo `Contextualize` sarebbe in grado di risolverla a runtime, oppure, ancora, tramite la proprietà `ViewContext` potrebbe risalire a tutte le informazioni riguardanti i file di risorse associati. Ci sono un paio di differenze rispetto a quanto visto con gli HTML localizer:

- Area di funzionamento: c'è una forte dipendenza da Razor e `IViewLocalizer` eredita da `IViewContextAware`, pertanto se utilizzato in un controller, verrebbe lanciata una eccezione perché non sarebbe possibile per il controller accedere a informazioni relative alla view. Il servizio `IHtmlLocalizer`, invece, poiché viene utilizzato in modo esplicito tramite i generics con un controller associato, può essere iniettato anche nel controller stesso al posto di un `IStringLocalizer`.

- File di risorse: sia per `IStringLocalizer` sia per `IHtmlLocalizer` è necessario un file di risorse per controller, mentre per `IViewLocalizer` è necessario un file di risorse specifico per ogni view.

Per quanto riguarda il secondo punto, ovvero per la gestione dei file di risorse, non c'è una soluzione più valida di un'altra: infatti, creare un file unico con tutte le risorse piuttosto che un file specifico per ogni view è solamente una scelta di business, che non va a influire sul funzionamento ma solo sulla manutenibilità. L'ordinamento, ovvero come e dove devono essere caricate le view localizzate e quale nome devono avere per essere viste dal sistema di localizzazione, dipende da quanto viene impostato nella proprietà `LanguageViewLocationExpander`.

Esempio 16.24

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddMvc()
```

```
        .AddViewLocalization(LanguageViewLocationExpanderFormat.Suffix);
    }
```

Questa proprietà, come viene mostrato nell'Esempio 16.24, per default assume il valore `Suffix` e pertanto può essere omessa dal metodo `AddViewLocalization` e si aspetta che il nome della culture, ovvero il tag IETF, sia già integrato nel nome della view (per esempio "Index.it-IT.cshtml"). Tutto il resto della struttura non cambia, quindi verrà mantenuta in linea di massima una cartella per controller con tutte le view associate all'interno. In base a quanto visto nell'introduzione di questo capitolo, sarebbe opportuno prevedere in ogni caso dei meccanismi di fallback sia per le culture padri (per esempio "it" nel caso dell'italiano) sia per la `InvariantCulture` (il classico `Index.cshtml`). Se invece di voler creare intere view localizzate, volessimo creare solo dei file di risorse associati alle view, allora un file verrebbe ricercato in `{root-progetto}/Views/{nome-controller}/{nome-view}.{tag-IETF}.resx`. Anche in questo caso vengono rispettati i meccanismi di fallback, quando la culture non venga recuperata al primo colpo e, inoltre, questi due sistemi possono lavorare assieme.

Nel caso in cui si preferisse una organizzazione in cartelle diversa da quella di default, bisognerebbe associare alla proprietà `LanguageViewLocationExpander` il valore `SubFolder`. L'unica differenza rispetto a quanto abbiamo appena detto per l'altro modello risulta nella definizione della struttura, che questa volta include i file in cartelle per tag IETF. Una vista "Index" potrebbe quindi essere ricercata in `Views/{nome-controller}/{tag-IETF}/Index.cshtml`, a meno che il percorso sia stato cambiato, come è illustrato nell'Esempio 16.15.

La localizzazione delle view va quasi a completare l'overview fatta, riguardante il supporto all'interno delle nostre applicazioni web, delle differenti culture. Nel caso in cui, per esempio, ci siano delle form di inserimento dati, però, potremmo trovare ancora qualche riferimento alla culture di default durante una verifica della validazione: per questo è necessario preoccuparsi anche della localizzazione delle data annotation.

Le data annotation

La validazione dei campi di una form, per esempio, viene fatta in due istanti temporali diversi: lato client, prima di effettuare il submit della form, e lato server, dove viene verificato che il modello sia valido. La validazione lato server è quella che affronteremo all'interno di questo capitolo poiché riguarda gli attributi di validazione, ovvero una parte delle data annotation, e di come questi attributi sfruttino il `ResourceManager` per recuperare le traduzioni dai file di risorse corrispondenti. La validazione lato client, invece, non sarà oggetto di questo capitolo poiché, nonostante sia ASP.NET a impostare gli attributi di validazione `data-*`, non viene gestita in modo diretto da ASP.NET Core e dipende principalmente dal sistema scelto, per esempio jQuery, che va a leggere ed elaborare quegli attributi.

Alcune delle data annotation più diffuse sono probabilmente quelle mostrate nell'Esempio 16.25, ovvero gli attributi `Required`, `MaxLength`, `Display` e `DataType`.

Esempio 16.25

```
[Required(ErrorMessage = "RequiredMessage")]
[MaxLength(50, ErrorMessage = "MaxLengthErrorMessage")]
[Display(Name = "FirstName")]
public string FirstName { get; set; }

[Required(ErrorMessage = "RequiredMessage")]
[DataType(DataType.EmailAddress, ErrorMessage = "EmailAddressErrorMessage")]
[Display(Name = "Email")]
public string Email { get; set; }
```

All'interno di tutte queste classi, c'è una proprietà che arriva dalla classe base `ValidationAttribute` chiamata `ErrorMessage`, che viene utilizzata e mostrata all'interno della form quando il modello inviato non è valido, cioè quando non rispetta gli attributi impostati. Questa proprietà è una stringa rappresentante un messaggio ma, qualora volessimo aggiungere la localizzazione, questo messaggio sarà la chiave utilizzata da ASP.NET Core per fare il lookup all'interno del file di risorse corrispondente al modello. La stessa cosa è valida per la proprietà `Name` relativa all'attributo `Display`, come viene mostrato nella Figura 16.9.

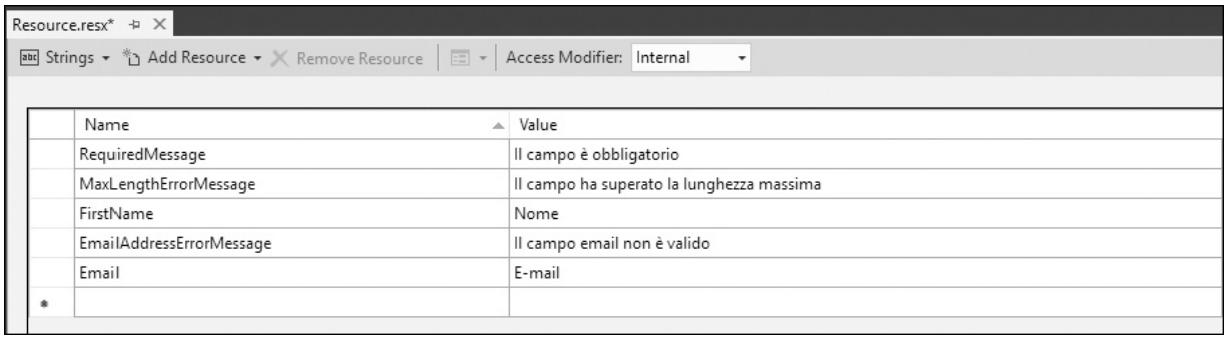


Figura 16.9 – Nel file di risorse per la localizzazione è anche possibile specificare i valori dei messaggi da assegnare alle data annotation.

Una volta creato il file di risorse associato al modello, la localizzazione continuerà a non funzionare poiché non è stato istruito il framework, quindi, come è mostrato nell'Esempio 16.26, registriamo la localizzazione per le data annotation nel file di startup.

Esempio 16.26

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddMvc()
    .AddDataAnnotationsLocalization();
}
```

Siccome, come per le view, anche questa modifica si riflette tra modello e view stesse, c'è una forte dipendenza e bisogna registrare il servizio insieme a MVC.

L'organizzazione strutturale dei file di risorse non cambia rispetto a quanto abbiamo visto già per la localizzazione di contenuti e view ma, dal momento che in questo caso si rischiano di creare molti file – almeno uno per modello – e considerando che probabilmente si vuole visualizzare lo stesso messaggio di errore per tutti gli attributi dello stesso tipo (per esempio, per tutti gli attributi `Required` si potrebbe voler mostrare "Il campo è obbligatorio"), potrebbe venir comodo creare una classe intermediaria, utilizzata dal `ResourceManager`, così che le risorse vengano recuperate da un solo generico file (per tag IETF).

Ancora una volta, grazie alla modularità di ASP.NET Core e al suo funzionamento a plug-in, è possibile specificare il provider che si occuperà di localizzare le risorse, come viene mostrato nell'Esempio 16.27.

Esempio 16.27

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddMvc()
    .AddDataAnnotationsLocalization(options =>
    {
      options.DataAnnotationLocalizerProvider = (type, factory) =>
      {
        return factory.Create(typeof(ErrorMessage));
      };
    });
}
```

Nell'esempio si può notare come il provider venga creato direttamente in linea, data la scarsa complessità; infatti, l'unica cosa che viene fatta è la creazione di una istanza tramite l'`IStringLocalizerFactory` di una classe `ErrorMessage` che è vuota ma che, come abbiamo anticipato, viene utilizzata dal `ResourceManager` per fare in modo che tutti i messaggi di errore (e in genere tutte le proprietà relative alle data annotation) vengano cercate nei file `ErrorMessage.{tag-IETF}.resx`.

Conclusioni

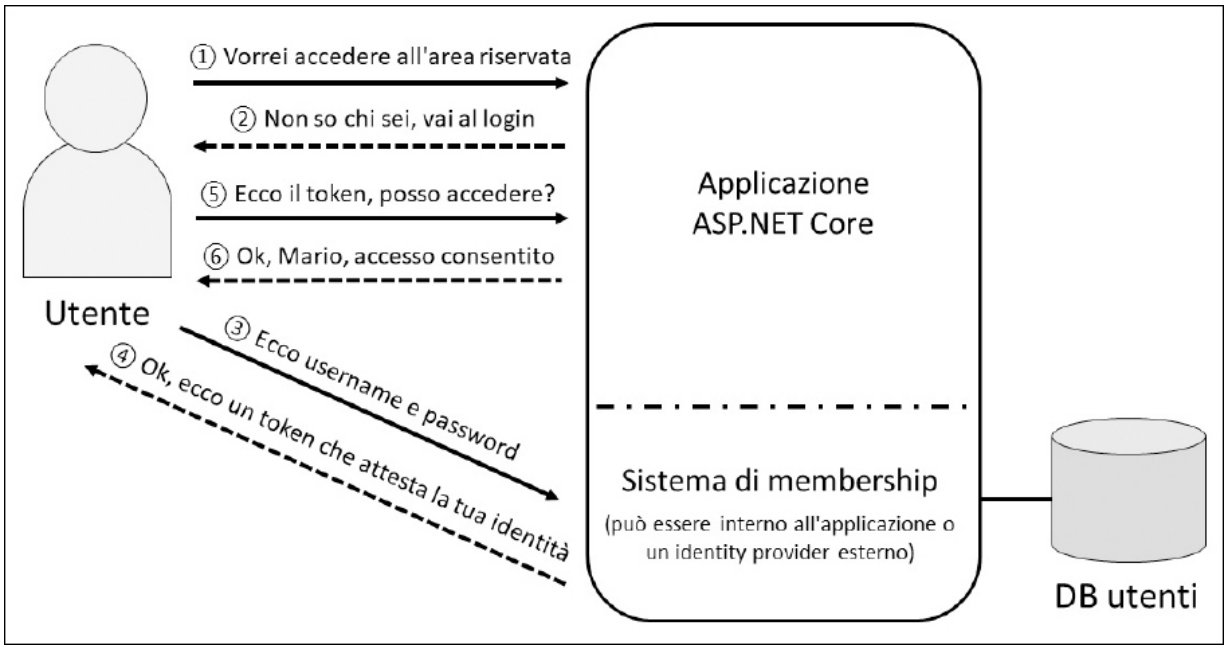
In questo capitolo abbiamo affrontato il tema dell'internazionalizzazione e abbiamo capito che è un approccio basato sulla globalizzazione, che avviene ancora prima dello sviluppo vero e proprio dell'applicazione, e sulla localizzazione, un processo che permette di rendere i contenuti adatti a una determinata lingua.

In termini di contenuti abbiamo visto quali sono le differenze tra applicare una localizzazione a un controller e a una view piuttosto che agli attributi di validazione dei modelli e si sono affrontate varie problematiche relative alle strategie di posizionamento dei file di risorse e alla loro modularità.

Utilizzando i middleware abbiamo costruito un'applicazione in grado di adattarsi ai vari agenti che effettuano le richieste, in modo che i contenuti serviti siano sempre coerenti in base al modello che è stato previsto, che sia tramite route piuttosto che tramite cookie oppure personalizzato.

La localizzazione è un aspetto importante delle applicazioni e non è da sottovalutare in un ambiente che ha bisogno di essere accessibile e fruibile da tutti. È un tema molto legato agli utenti finali che utilizzeranno l'applicazione e pertanto, come vedremo nel prossimo capitolo, è bene capire chi sono gli utenti che stanno utilizzando l'applicazione e quali ruoli hanno, per fornire loro contenuti sempre più personalizzati.

- ❑
- ❑
- ❑

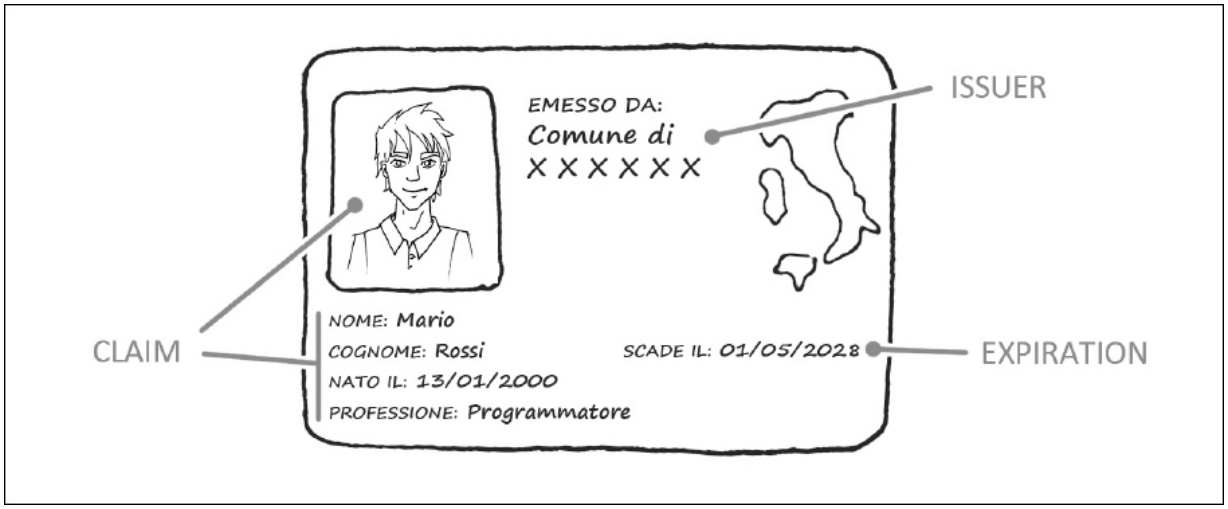


La fase di login

- ❑

- ❑

Un’identità formata da claim

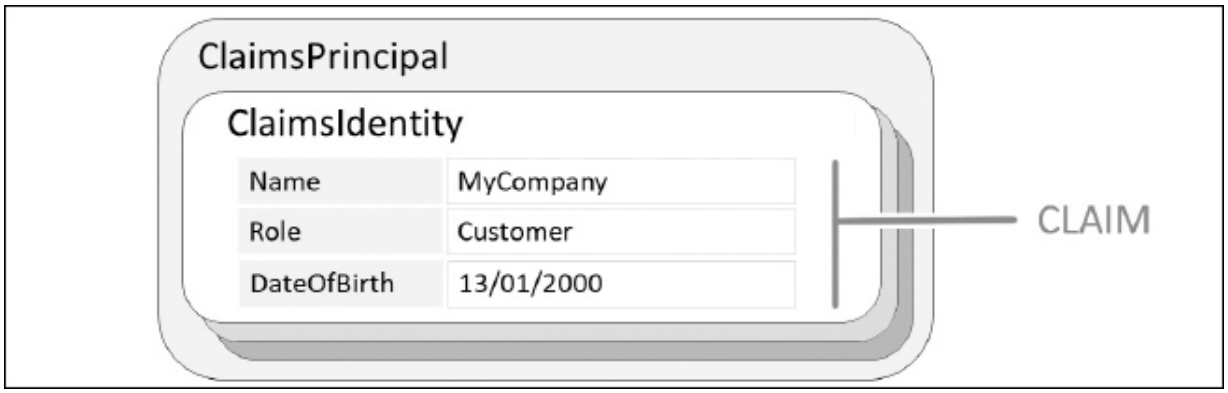


La fase di autenticazione

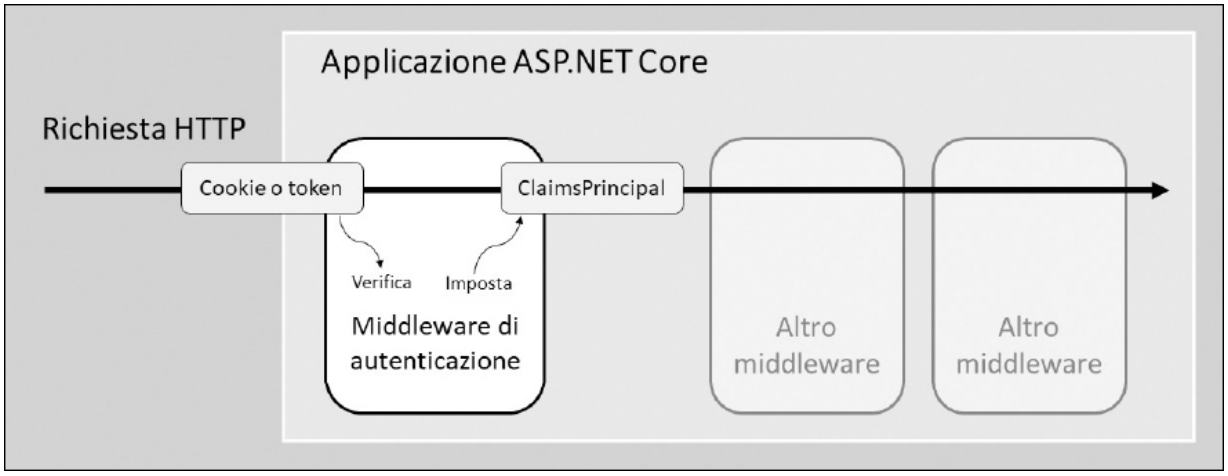
`ClaimsIdentity`

`ClaimsIdentity`

`ClaimsPrincipal`



ClaimsPrincipal

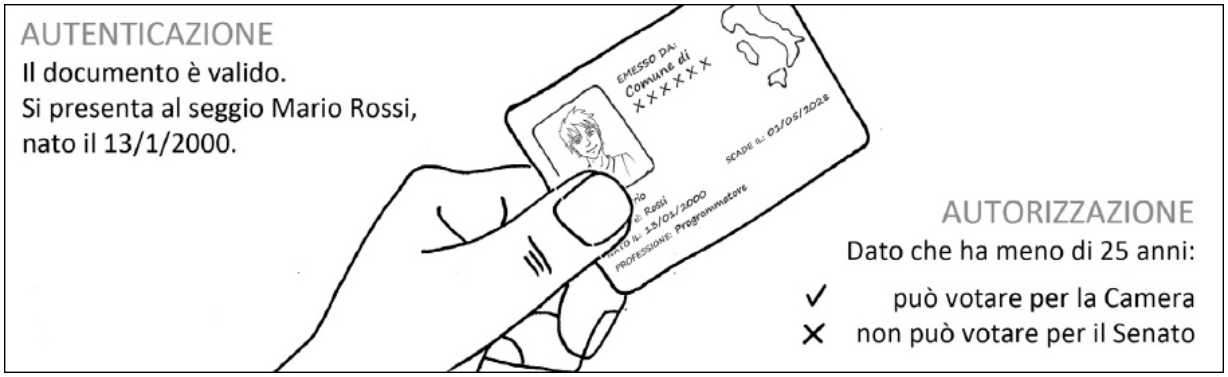


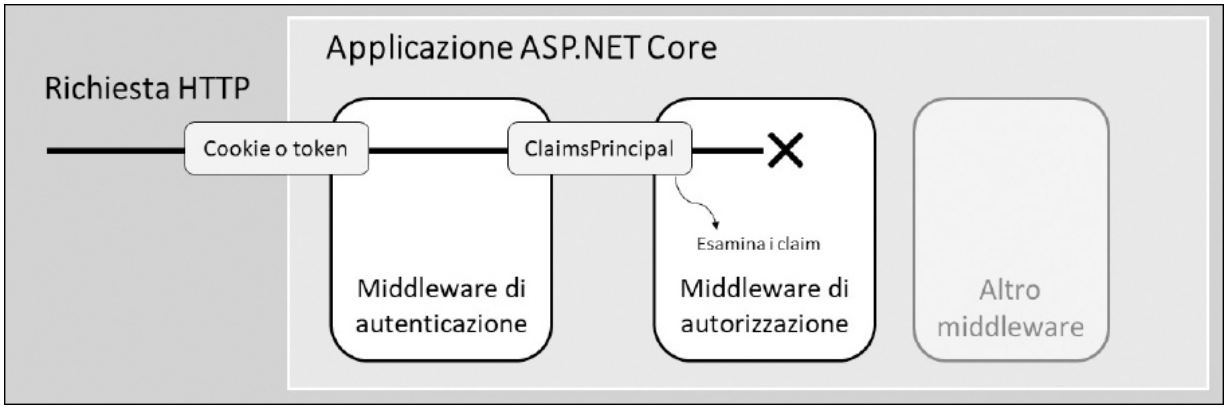
ClaimsPrincipal

HttpContext.User

La fase di autorizzazione

ClaimsPrincipal






```
AuthorizeAttribute
```

Membership con ASP.NET Core Identity

-

  UserManager<TUser>

-

mvc

webapi

```
dotnet new mvc --auth Individual
```

--auth Individual

--use-local-db

appsettings.json

ConfigureServices

Startup

```
services.AddDefaultIdentity<IdentityUser>()
        .AddEntityFrameworkStores<ApplicationDbContext>();
```

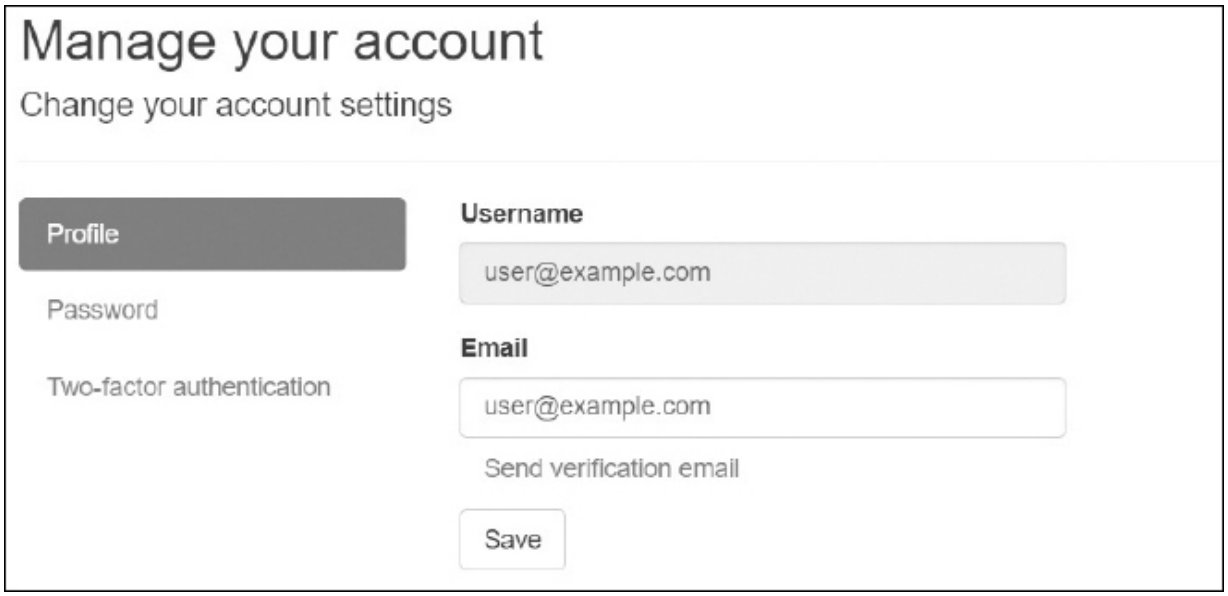
AddDefaultIdentity

IdentityUser

AddEntityFrameworkStores

DbContext

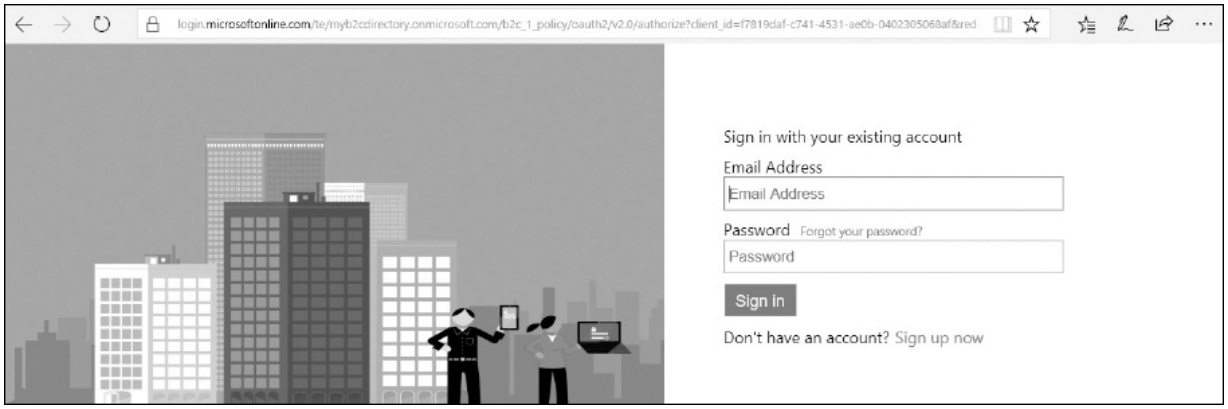
http://aspit.co/bpl



AddDefaultIdentity

AddIdentityCore

Membership con Azure Active Directory B2C



http://aspit.co/bmz

```
dotnet new mvc --auth IndividualB2C
```

Startup

appsettings.json

-
-

Configure Startup

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
  //Agiungiamo il middleware di autenticazione di ASP.NET Core
  app.UseAuthentication();

  //Qui configurazione di altri middleware
}
```

ClaimsPrincipal

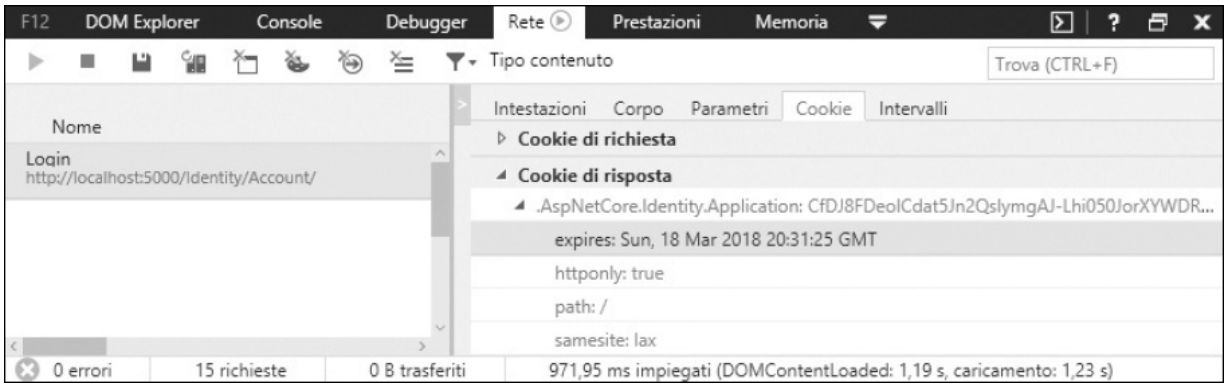
Autenticazione su cookie con ASP.NET Core Identity

ConfigureApplicationCookie

ExpireTimeSpan

```
public void ConfigureServices(IServiceCollection services)
{
  services.ConfigureApplicationCookie(options =>
  {
    //Sliding expiration di un giorno
    options.ExpireTimeSpan = TimeSpan.FromDays(1);
    //Abilitiamo questo se vogliamo impostare una absolute expiration
    //options.SlidingExpiration = false;
  });
  //Qui configurazione di altri servizi
}
```

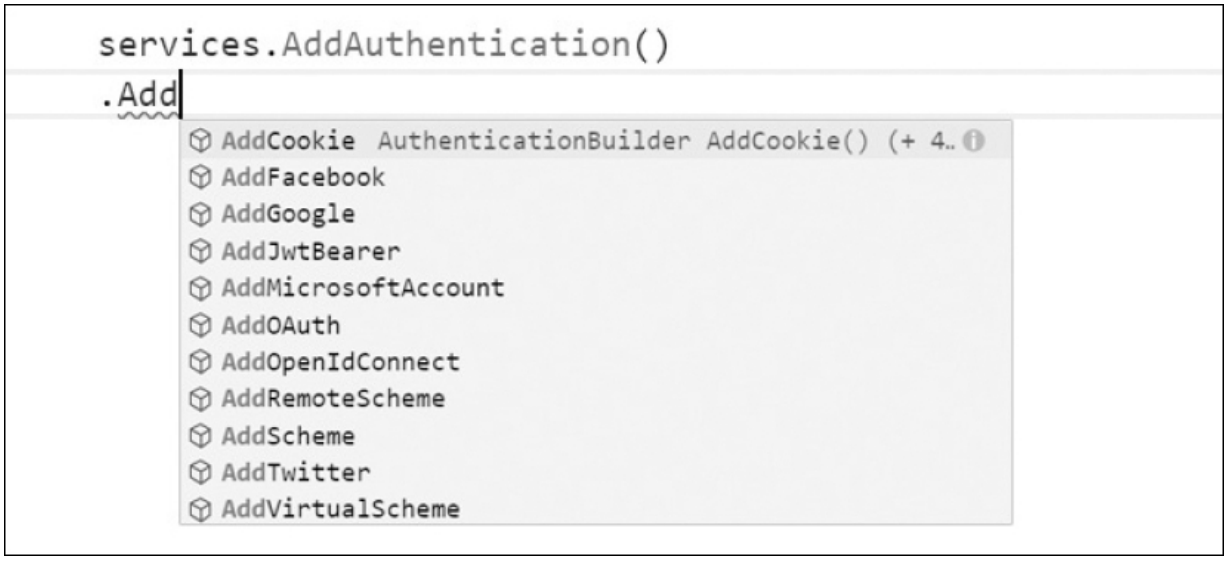
SlidingExpiration false



Autenticazione su cookie senza ASP.NET Core Identity

`ConfigureServices` `Startup`

AddAuthentication



AddCookie

```
public void ConfigureServices(IServiceCollection services)
{
  //Usiamo AddAuthentication per poi configurare gli scheme supportati
  services.AddAuthentication(
    defaultScheme: CookieAuthenticationDefaults.AuthenticationScheme
  )
  //Aggiungiamo lo scheme della cookie authentication
  .AddCookie(CookieAuthenticationDefaults.AuthenticationScheme, options =>
  {
    //Modifichiamo qui le opzioni di emissione dei cookie
    options.ExpireTimeSpan = TimeSpan.FromDays(1);
  });
  //Qui registrazione di altri servizi
}
```

HttpContext.SignInAsync

```
[HttpPost]
public async Task<IActionResult> Login(LoginModel login, string redirectUrl)
{
  //Verifichiamo se esiste un utente con le credenziali fornite
  //Il metodo GetUserByCredentials contiene la logica personalizzata
  //per tale verifica. Lo user è anch´esso un nostro oggetto personalizzato
  var user = await GetUserByCredentials(login.Username, login.Password);
  if (user == null) {
    //Nessun utente trovato, vuol dire che le credenziali non erano valide
    ViewBag.Error = ″Credentials are not valid!″;
    return View();
  }

  //Creiamo una ClaimsIdentity e aggiungiamo i claim dell´utente loggato
  var claimsIdentity = new ClaimsIdentity(
    authenticationType: CookieAuthenticationDefaults.AuthenticationScheme
  );
  claimsIdentity.AddClaim(new Claim(ClaimTypes.Name, user.Username));
  claimsIdentity.AddClaim(new Claim(ClaimTypes.Role, user.Role));

  //Incapsuliamo tutto in un ClaimsPrincipal
  var claimsPrincipal = new ClaimsPrincipal(claimsIdentity);
  //Emettiamo il cookie di autenticazione fornendo delle opzioni
  var authProperties = new AuthenticationProperties {
    //Se l´utente vuole restare loggato, il cookie è persistente
    IsPersistent = login.RememberMe
  };
  await HttpContext.SignInAsync(
    scheme: CookieAuthenticationDefaults.AuthenticationScheme,
    principal: claimsPrincipal,
    properties: authProperties);

  //Login effettuato: indirizziamo l´utente alla pagina di provenienza
  return Redirect(redirectUrl ?? ″/″);
}
```

IsPersistent

AuthenticationProperties
HttpContext.SignInAsync

HttpContext.SignOutAsync

```
public async Task<IActionResult> Logout() {
  await HttpContext.SignOutAsync(
    scheme: CookieAuthenticationDefaults.AuthenticationScheme
  );
  return Redirect("/");
}
```

custom-cookie-auth

Autenticazione su token JWT

http://aspit.co/bnt

AddJwtBearer

```
services.AddAuthentication(opts =>
{
  opts.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
}).AddJwtBearer(opts =>
{
  opts.TokenValidationParameters = new TokenValidationParameters
  {
    ValidateIssuer = true,
    ValidateAudience = true,
    ValidateIssuerSigningKey = true,
    ValidIssuer = ″Issuer″,
    ValidAudience = ″Audience″,
    IssuerSigningKey = new SymmetricSecurityKey(
      Encoding.UTF8.GetBytes(″SecretKey″)
    ),
    //Tolleranza sulla data di scadenza del token
    ClockSkew = TimeSpan.Zero
  };
});
```

```
private string CreateTokenForIdentity(IEnumerable<Claim> claims)
{
  var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(″SecretKey″));
  var cred = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);
  var token = new JwtSecurityToken(
    issuer: ″Issuer″,
    audience: ″Audience″,
    claims: claims,
    expires: DateTime.Now.AddMinutes(20),
    signingCredentials: cred
  );
  return new JwtSecurityTokenHandler().WriteToken(token);
}
```

Add

ui-and-webapi-auth

Login con i social network

http://aspit.co/bne

```
services.AddAuthentication()
.AddMicrosoftAccount(options => {
  options.ClientId = ″myid″;
  options.ClientSecret = ″mysecret″;
})
.AddFacebook(options => {
  options.AppId = ″myid″;
  options.AppSecret = ″mysecret″;
})
.AddGoogle(options => {
  options.ClientId = ″myid″;
  options.ClientSecret = ″mysecret″;
})
.AddTwitter(options => {
  options.ConsumerKey = ″mykey″;
  options.ConsumerSecret = ″mysecret″;
});
```

Login su provider esterni con OAuth 2.0 e OpenID Connect

AddOAuth

AddOpenIdConnect

```
services.AddAuthentication()
.AddOAuth("SchemeName", options => {
  //Configurazione specifica
})
.AddOpenIdConnect(options => {
  //Configurazione specifica
});
```

Come guida all'utilizzo degli extension method `AddOAuth` *e* `AddOpenIdConnect` *Microsoft ha pubblicato delle applicazioni dimostrative nel repository GitHub di ASP.NET Core Identity, raggiungibile all'indirizzo: http://aspit.co/bnh. In particolar modo, sono rilevanti le classi Startup degli esempi SocialSample e OpenIdConnectSample per avere una panoramica dei valori di configurazione coinvolti.*

http://aspit.co/bni

Autenticazione Windows

Program

```
public static IWebHost BuildWebHost(string[] args) =>
  WebHost.CreateDefaultBuilder(args)
  .UseHttpSys(options =>
  {
  options.Authentication.Schemes =
    AuthenticationSchemes.NTLM |  AuthenticationSchemes.Negotiate;
  //Se vogliamo che l´utente risulti loggato in qualsiasi pagina
  //options.Authentication.AllowAnonymous = false;
  })
  .UseStartup<Startup>()
  .Build();
```

AddAuthentication

```
services.AddAuthentication(HttpSysDefaults.AuthenticationScheme);
```

http://nomemacchina

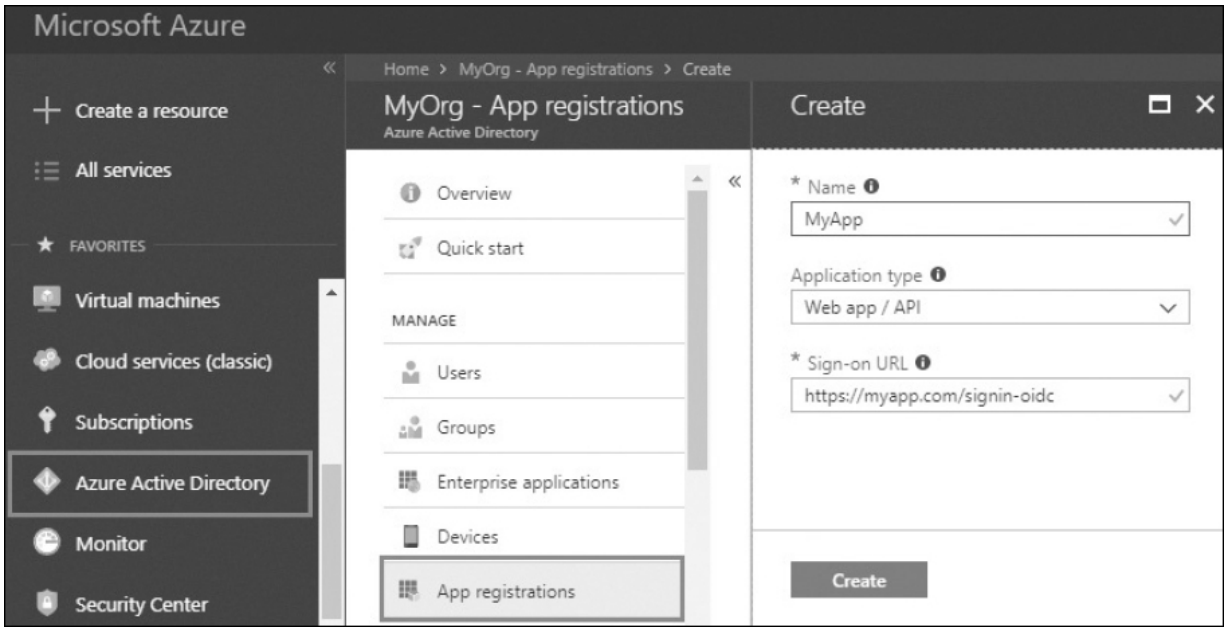
WindowsIdentity

ClaimsIdentity

RunImpersonated

```
var identity = HttpContext.User.Identity as WindowsIdentity;
WindowsIdentity.RunImpersonated(identity.AccessToken, () =>
{
  //Qui accesso a risorse con privilegi dell´utente: es. lettura di un file
});
```

Autenticazione con Azure Active Directory


Microsoft Azure
Home > MyOrg - App registrations > Create
Create a resource
All services
FAVORITES
Virtual machines
Cloud services (classic)
Subscriptions
Azure Active Directory
Monitor
Security Center
MyOrg - App registrations
Azure Active Directory
Overview
Quick start
MANAGE
Users
Groups
Enterprise applications
Devices
App registrations
Create
* Name
MyApp
Application type
Web app / API
* Sign-on URL
https://myapp.com/signin-oidc
Create

❑

❑

AddAzureAd

```
services.AddAuthentication(opts => {
  opts.DefaultScheme = CookieAuthenticationDefaults.AuthenticationScheme;
  opts.DefaultChallengeScheme = OpenIdConnectDefaults.AuthenticationScheme;
})
.AddAzureAd(opts => {
  opts.Instance = ″https://login.microsoftonline.com/″;
  opts.Domain = ″myorg.onmicrosoft.com″;
  opts.TenantId = ″aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee″;
  opts.ClientId = ″11111111-2222-3333-4444-555555555555″;
  opts.CallbackPath = ″/signin-oidc″;
})
.AddCookie();
```

```
[HttpGet]
public IActionResult SignIn()
{
  //Url di ridirezione a login avvenuto
  var redirectUrl = Url.Action(″Index″, ″Home″);
  return Challenge(
    new AuthenticationProperties { RedirectUri = redirectUrl },
    OpenIdConnectDefaults.AuthenticationScheme);
}
```

http://aspit.co/bnj

http://aspit.co/bnk

AuthorizeAttribute

L’attributo AuthorizeAttribute

AuthorizeAttribute

```
[Authorize]
public class AccountController : Controller
{
  //Qui sono definite le action del controller
}
```

AuthorizeFilter

AddMvc

ConfigureServices Startup

```
services.AddMvc(options => {
  var policy = new AuthorizationPolicyBuilder()
                   .RequireAuthenticatedUser()
                   .Build();
  var filter = new AuthorizeFilter(policy);
  //Ogni controller e ogni action saranno inaccessibili agli utenti anonimi
  options.Filters.Add(filter);
});
```

L'attributo AllowAnonymousAttribute

AllowAnonymousAttribute ,

```
[Authorize]
public class AccountController : Controller {
  [HttpGet, AllowAnonymous]
  public async Task<IActionResult> Login() {
    // Mostriamo la view di login all´utente
  }
  [HttpPost, AllowAnonymous]
  public async Task<IActionResult> Login(LoginViewModel model) {
    // Qui verifichiamo le credenziali fornite
  }
  //Qui altre action protette
}
```

AuthorizeAttribute

Autorizzazione in base al ruolo

AuthorizeAttribute

AuthorizeAttribute

Roles

IndexAll

```
[Authorize]
public class CustomerController : Controller {
  [Authorize(Roles = ″Administrator,PowerUser″)]
  public async Task<IActionResult> IndexAll() {
    //Solo gli utenti con ruolo Administrator o con ruolo PowerUser
    //potranno visualizzare l´elenco completo dei clienti
  }
}
```

ClaimsIdentity ClaimTypes.Role

Usare le policy per criteri avanzati di autorizzazione

AddAuthorization

ConfigureServices Startup

```
services.AddAuthorization(options => {
  options.AddPolicy("JustForCustomer1", policy => {
    //Imponiamo la presenza del claim Email
    policy.RequireClaim(ClaimTypes.Email);
    //Imponiamo che il claim "Tenant" sia uguale a "Customer1"
    policy.RequireClaim("Tenant", "Customer1");
  });
});

//Qui registrazione di altri servizi
```

RequireClaim

RequireRole RequireUserName

RequireAuthenticatedUser

Policy AuthorizeAttribute

```
public class ReportController : Controller
{
  [Authorize(Policy = "JustForCustomer1")]
  public async Task<IActionResult> SendCustomizedReportViaEmail() {
    //Invia un report all'indirizzo email dell'utente
  }
}
```

Policy con logica di autorizzazione personalizzata

```
services.AddAuthorization(options => {
  options.AddPolicy("VipCustomers", policy => {
    //Aggiungiamo il requirement di un importo minimo acquistato
    policy.Requirements.Add(
      new MinimumPurchaseRequirement(minimum: 1000)
    );
  });
});
```

ClaimsIdentity

IAuthorizationRequirement

```
public class MinimumPurchaseRequirement : IAuthorizationRequirement
{
  public MinimumPurchaseRequirement(decimal minimum) {
    Minimum = minimum;
  }
  public decimal Minimum { get; }
}
```

AuthorizationHandler<TRequirement>

```
public class MinimumPurchaseAuthorizationHandler :
  AuthorizationHandler<MinimumPurchaseRequirement>
{
  protected override async Task HandleRequirementAsync(
    AuthorizationHandlerContext context,
    MinimumPurchaseRequirement requirement)
  {
    //Recuperiamo dal database il valore speso fino a oggi dall´utente
    decimal purchased = await GetAmount(context.User.Identity.Name);
    //Lo confrontiamo con quello minimo indicato nel requirement
    if (purchased < requirement.Minimum)
    {
      //Non ha raggiunto il minimo, l´autorizzazione fallisce
      context.Fail();
    }
    //Altrimenti, l´autorizzazione ha successo
    context.Succeed(requirement);
  }
}
```

HandleRequirementAsync

Succeed Fail

AuthorizationHandlerContext

AuthorizationHandler

DbContext

AuthorizationHandler

ClaimsIdentity

```
public void ConfigureServices(IServiceCollection services) {
  services.AddScoped<IAuthorizationHandler,
        MinimumPurchaseAuthorizationHandler>();
}
```

AddScoped

AddSingleton

Autorizzazione in base all'authentication scheme

AuthorizeAttribute

AuthenticationSchemes

```
[Authorize(Roles = "Crm")]
public class SaleController : Controller {
  [Authorize(AuthenticationSchemes = JwtBearerDefaults.AuthenticationScheme)]
  public async Task<IActionResult> GetRecentOrders() {
    //logica
  }
}
```

AuthorizeAttribute

Autorizzazione on demand con IAutorizationService

AuthorizeAttribute

IAuthorizationService AuthorizeAsync

```
public async Task<IActionResult> DeleteOrder(
  int orderId, [FromServices] IAuthorizationService authService)
{
  //Usiamo l´orderId che ci viene valorizzato dal Model Binder
  Order order = await ctx.Orders.FindAsync(orderId);

  //Forniamo l´utente, la risorsa da autorizzare e la policy
  AuthorizationResult result =
        await authService.AuthorizeAsync(User, order, ″SenderOfOrder″);
  if (result.Succeeded) {
    //Eliminiamo l´ordine ma solo se l´autorizzazione ha avuto successo
    ctx.Orders.Remove(order);
    await ctx.SaveChangesAsync();
  }
  return RedirectToAction(nameof(Index));
}
```

customer-authorization

Aggiungere proprietà personalizzate al profilo dell'utente

IdentityUser

ApplicationUser

```
public class ApplicationUser : IdentityUser
{
  //Aggiungiamo una proprietà personalizzata
  public string FiscalCode { get; set; }
}
```

ApplicationUser

ApplicationDbContext
IdentityContext<TUser>

```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
  //Eventuale mapping aggiuntivo nel metodo OnModelCreating
}
```

```
dotnet ef migrations add ″FiscalCode added″
dotnet ef database update
```

ConfigureServices Startup

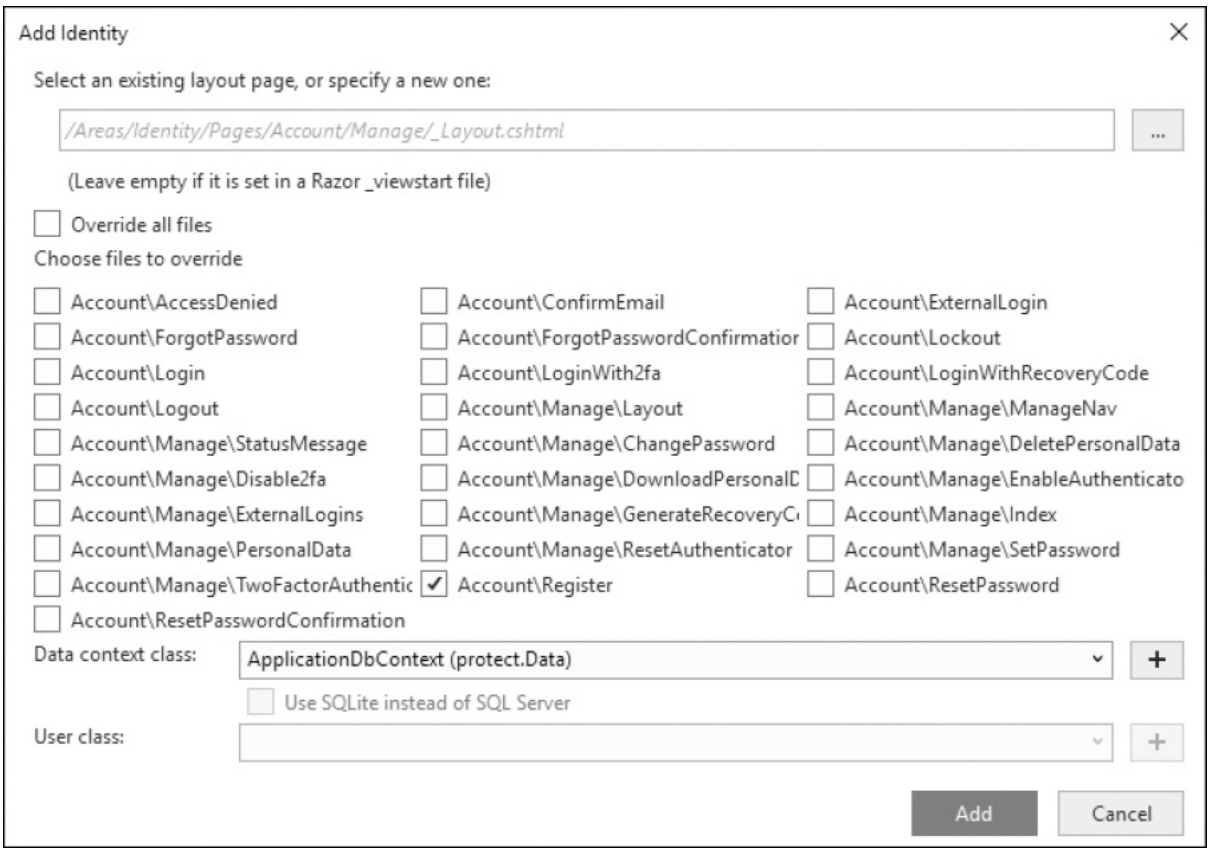
```
services.AddDefaultIdentity<ApplicationUser>()
        .AddEntityFrameworkStores<ApplicationDbContext>();
```

UserManager<IdentityUser>

UserManager<ApplicationUser>
ApplicationUser

Personalizzare la pagina di registrazione

Add -> New scaffolded item -> Identity



.cshtml
.cshtml.cs
Register.cshtml

```
<div class="form-group">
  <label asp-for="Input.FiscalCode"></label>
  <input asp-for="Input.FiscalCode" class="form-control" />
  <span asp-validation-for="Input.FiscalCode" class="text-danger"></span>
</div>
```

FiscalCode

```
[Required]
[Display(Name = "Codice fiscale")]
public string FiscalCode { get; set; }
```

[Required]

FiscalCode
ApplicationUser

OnPostAsync

```
public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
  if (ModelState.IsValid)
  {
    var user = new ApplicationUser
```

```
      {
        UserName = Input.Email,
        Email = Input.Email,
        //Copiamo il codice fiscale sull´ApplicationUser
        FiscalCode = Input.FiscalCode
      };

      var result = await _userManager.CreateAsync(user, Input.Password);
      //...
    }
    //...
  }
```

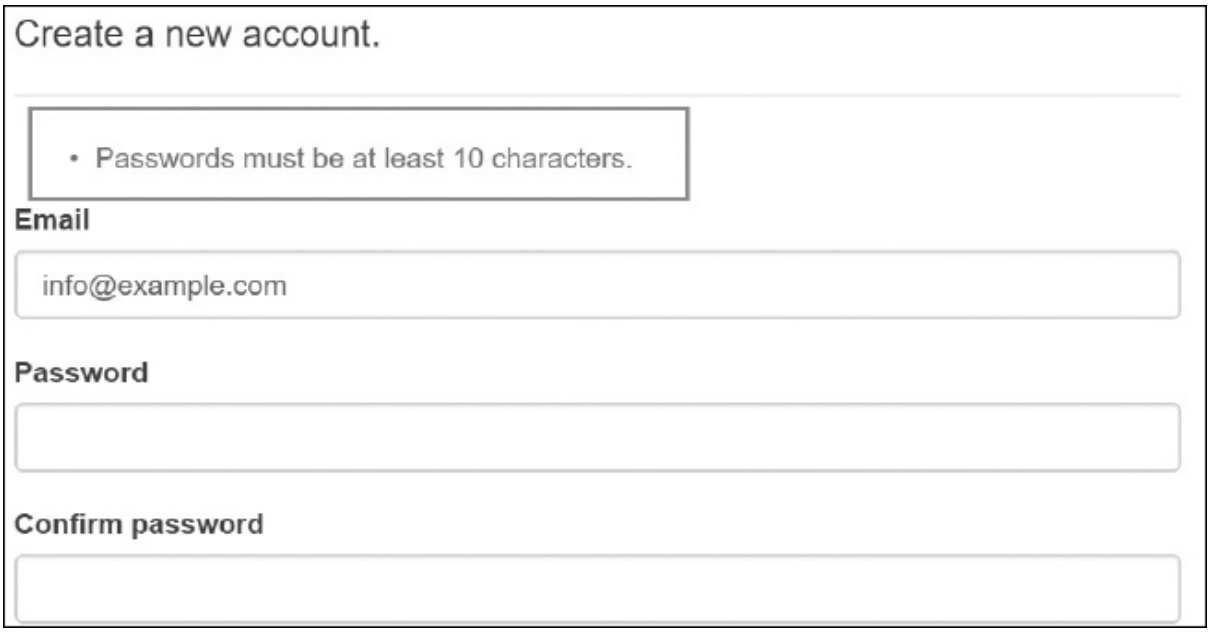
PersonalData.cshtml

http://aspit.co/bos

Criteri di sicurezza della password

```
services.AddDefaultIdentity<ApplicationUser>(options => {
  options.Password.RequiredLength = 10;
  options.Password.RequireDigit = false;
  options.Password.RequireNonAlphanumeric = false;
})
.AddEntityFrameworkStores<ApplicationDbContext>();
```

PasswordHasher<TUser>




```
//MyHasher è una implementazione di IPasswordHasher<ApplicationUser>
services.AddTransient<IPasswordHasher<ApplicationUser>, MyHasher>();
```

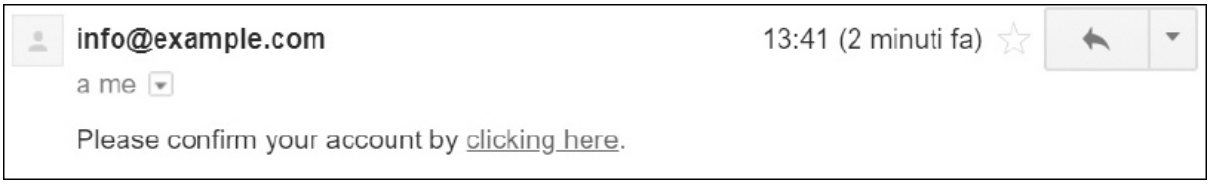
Chiedere la conferma di registrazione

```
services.AddDefaultIdentity<ApplicationUser>(options => {
  options.SignIn.RequireConfirmedEmail = true;
})
.AddEntityFrameworkStores<ApplicationDbContext>();
```

IEmailSender

SmtpClient

```
services.AddSingleton<IEmailSender, EmailSender>();
```



Register.cshtml.cs

```
//await signInManager.SignInAsync(user, isPersistent: false);
```

Reimpostazione della password

EmailTokenProvider

AuthenticatorTokenProvider

Two-factor authentication

Manage your account

Change your account settings

Profile

Password

Two-factor authentication

Personal data

Configure authenticator app

To use an authenticator app go through the following steps:

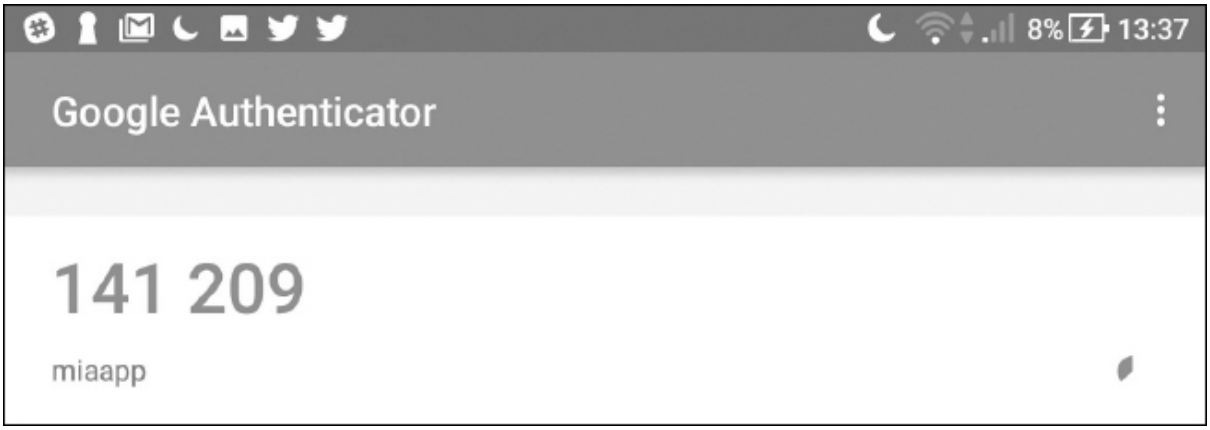
1. Download a two-factor authenticator app like Microsoft Authenticator for Windows Phone, Android and iOS or Google Authenticator for Android and iOS.
2. Scan the QR Code or enter this key `5y2c zgl4 cdli o226 x5tx clsk 6qxh c42g` into your two factor authenticator app. Spaces and casing do not matter.

   To enable QR code generation please read our documentation.

3. Once you have scanned the QR code or input the key above, your two factor authentication app will provide you with a unique code. Enter the code in the confirmation box below.

   Verification Code

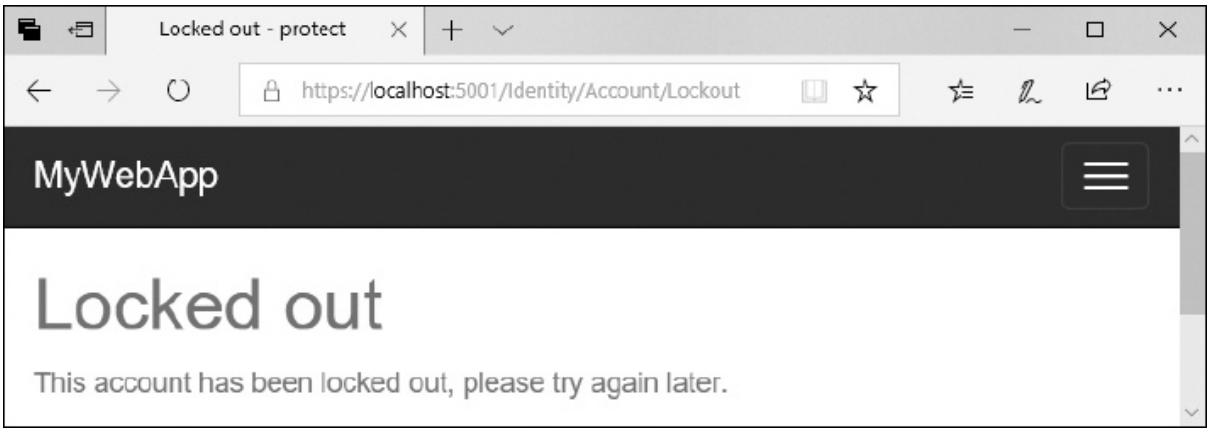
   Verify



Protezione dai tentativi di accesso brute force

```
SignInManager
```

```
services.AddDefaultIdentity<ApplicationUser>(options => {
  //Il lockout è abilitato per default
  //options.Lockout.AllowedForNewUsers = true;

  //Impostiamo durata del blocco e tentantivi massimi
  options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromHours(6);
  options.Lockout.MaxFailedAccessAttempts = 3;
})
.AddEntityFrameworkStores<ApplicationDbContext>();
```



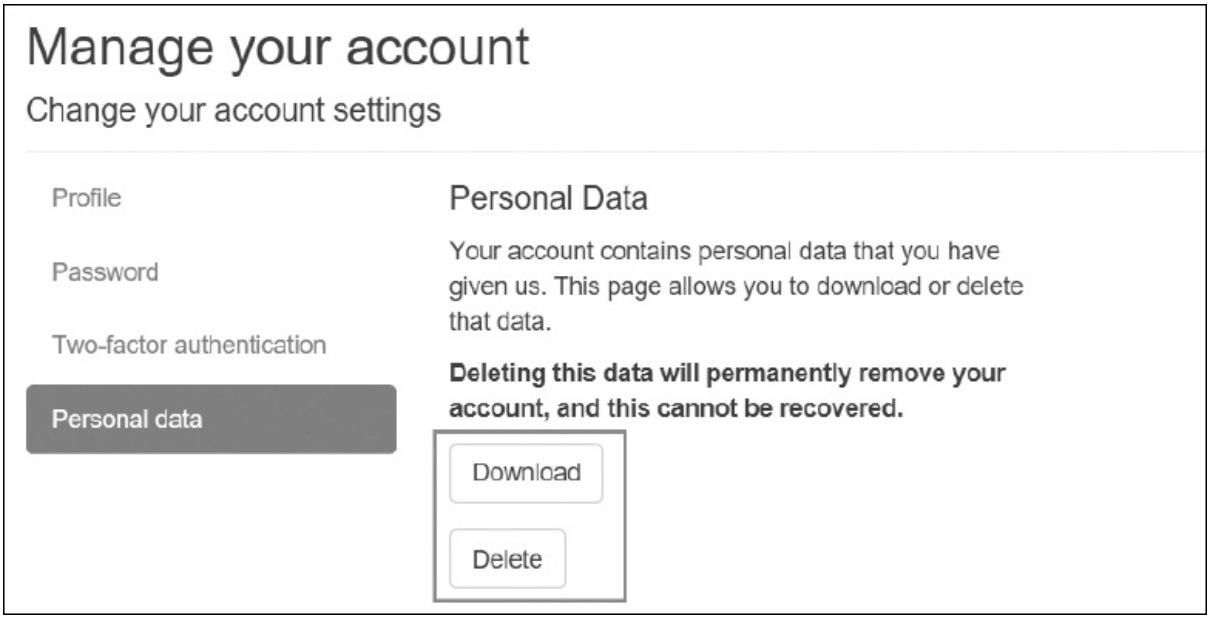
Sign-out remoto

SecurityStamp

SecurityStamp

```
services.Configure<SecurityStampValidatorOptions>(options => {
  //Possiamo impostare TimeSpan.Zero per controllarlo ad ogni richiesta
  options.ValidationInterval = TimeSpan.FromMinutes(1);
});
```

Controllo dei dati da parte dell'utente per il GDPR



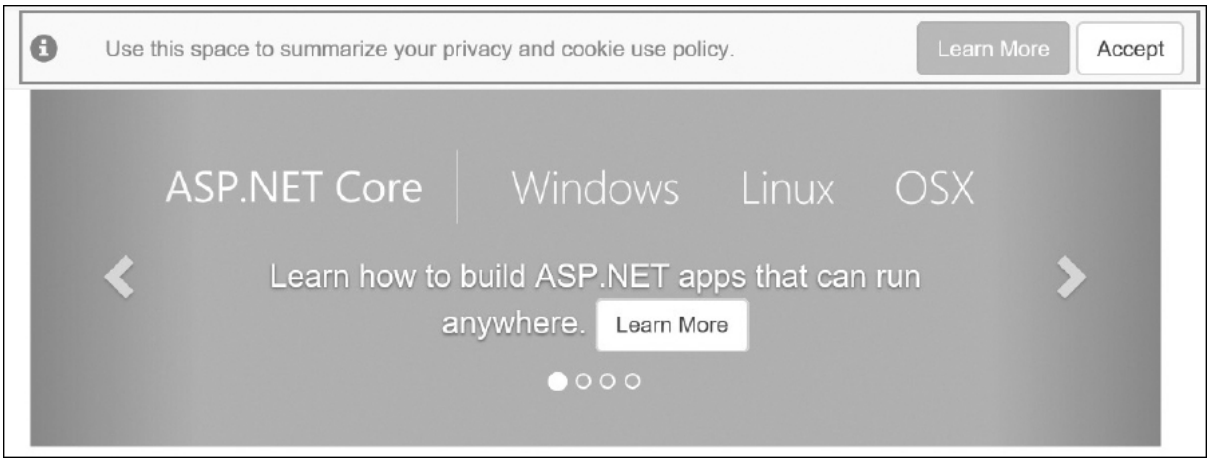
ApplicationUser
DownloadPersonalData.cshtml DeletePersonalData.cshtml

http://aspit.co/bpb

Rispetto della normativa europea sui cookie

CookiePolicyMiddleware

/Views/Shared/_CookieConsentPartial.cshtml



CookieOptions

```
var cookieOptions = new CookieOptions {
  Expires = DateTimeOffset.Now.AddMonths(1),
  IsEssential = false
};
HttpContext.Response.Cookies.Append(″TrackId″, ″128432″, cookieOptions);
```

UserManager<TUser>

-
- ClaimsIdentity
-
-

UserManager<TUser>

```
public class UserManagementController : Controller
{
  private readonly UserManager<ApplicationUser> userManager;
  public UserManagementController(UserManager<ApplicationUser> userManager)
  {
    this.userManager = userManager;
  }
  //Qui le action che usereanno lo userManager per compiere operazioni
}
```

TUser
ApplicationUser
UserManager

UserManager

Creare un utente programmaticamente

CreateAsync
UserManager<TUser>

```
var user = new ApplicationUser {
  Email = "user1@example.com",
  UserName = "user1@example.com",
  EmailConfirmed = true
};
await userManager.CreateAsync(user, "Password1!");
```

ApplicationUser

Interrogare il database degli utenti locali

Users IQueryable<TUser>

```
var users = await userManager.Users
  .Where(user => user.Email.EndsWith("@example.com"))
  .OrderBy(user => user.Email)
  .ToListAsync();
```

UserManager

FindByNameAsync FindByEmailAsync

FindByIdAsync

Assegnare claim e modificare i dati di login dell'utente

UserManager
AddClaimAsync
UpdateAsync

```
//Estraggo l´utente in base alla sua e-mail
var user = await userManager.FindByEmailAsync(˝user1@example.com˝);

//Creo un claim e glielo associo
var claim = new Claim(ClaimTypes.Role, ˝Customer˝);
await userManager.AddClaimAsync(user, claim);

//Aggiorno la password
user.PasswordHash = userManager.PasswordHasher.HashPassword(user, ˝New1!˝);
await userManager.UpdateAsync(user);
```

RemoveClaimAsync UserManager

user-administration

Proteggere l'applicazione con un certificato SSL

Example Domain
Not secure
http://www.example.com
Example Domain

Program

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
  WebHost.CreateDefaultBuilder(args)
  .UseKestrel((context, options) => {
  options.Listen(IPAddress.Loopback, 5000, listenOptions => {
    listenOptions.UseHttps(httpsOptions =>
    {
      //Possiamo leggere il certificato da file
      var cert = new X509Certificate2(@"cert.pfx", "password");
      //Oppure dal certificate store (su Windows)
      //var cert = CertificateLoader.LoadFromStoreCert(
        "localhost", "My", StoreLocation.CurrentUser, allowInvalid: false);
      //Impostiamo un singolo certificato
      httpsOptions.ServerCertificate = localhostCert;
      //Oppure usiamo SNI per selezionare il certificato in base all´host
      //httpsOptions.ServerCertificateSelector = (context, nomeHost) => {
      // return localhostCert;
      //};
    });
  });
})
.UseStartup<Startup>();
```

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
  WebHost.CreateDefaultBuilder(args)
  .UseHttpSys(options => {
    options.UrlPrefixes.Add("https://www.example.com:443");
    options.UrlPrefixes.Add("https://tenant1.example.com:443");
    //...
  })
  .UseStartup<Startup>();
```

netsh http add sslcert http://aspit.co/bol

http://aspit.co/bok

Forzare il traffico su HTTPS

[RequireHttps] ConfigureServices Startup

```
services.AddMvc(options => {
  options.Filters.Add(new RequireHttpsAttribute());
  //Decommentiamo questa riga se vogliamo indicare una porta diversa
  //options.SslPort = 443;
});
```

HttpsRedirectionMiddleware Configure Startup

```
app.UseHttpsRedirection();
```

http://

```
//Produrrà l´intestazione che indica al browser di fare richieste su HTTPS
//Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
app.UseHsts();
```

AddHsts ConfigureServices Startup

Cross Site Scripting (XSS)

HtmlEncoder

```
@{
  //In un´applicazione reale questo valore arriva dal database
  var contenuto = ″<iframe src=´http://www.sitomalevolo.com´></iframe>″;
}
<h4>L´utente ha scritto:</h4> <p>@contenuto</p>
```

iframe

< >

< >

L'utente ha scritto:

<iframe src='http://www.sitomalevolo.com'></iframe>

@Html.Raw HtmlEncoder

```
@Html.Raw(″<strong>Questo testo è rappresentato in grassetto</strong>″)
```

HtmlEncoder

JavaScriptEncoder UrlEncoder

Encode

Cross Site Request Forgery (CSRF o XSRF)

```
<form action=″http://example.com/account″ method=″post″>
  <input type=″hidden″ name=″Importo″ value=″10000″>
```

```
    <input type=″hidden″ name=″Iban″ value=″IT80R111111111111111111″>
    <input type=″submit″ name=″InviaBonifico″ value=″Ricevi il premio!″>
  </form>
```

RequestVerificationToken

<form>

```
◢ <form method="post">
    <input type="submit" value="Invia" />
    <input name="__RequestVerificationToken" type="hidden" value="CfDJ8HYl1dua..." />
  </form>
```

[AutoValidateAntiforgeryToken]

```
services.AddMvc(options => {
  options.Filters.Add(new AutoValidateAntiforgeryTokenAttribute());
});
```

POST PUT DELETE
GET

Open Redirect

returnUrl

```
https://example.com/login?returnUrl=https%3A%2F%2www.exxxample.com
```

returnUrl

LocalRedirect

```
[HttpPost]
public async Task<IActionResult> Login(LoginModel login, string returnUrl)
{
  if (ModelState.IsValid)
  {
    //Procedura di verifica delle credenziali personalizzata
    if (await Verifica(loginModel.Username, loginModel.Password))
    {
      //Usiamo LocalRedirect per evitare gli attacchi Open Redirect
      return LocalRedirect(returnUrl);
    }
  }
  return View(loginViewModel);
}
```

LocalRedirect returnUrl Url.IsLocalUrl false

Cross-Origin Resource Sharing (CORS)

AddCors ConfigureServices Startup

```
services.AddCors(options => {
  options.AddPolicy("AllowOrigins", builder => {
    //Consentiamo richieste con qualsiasi intestazione e metodo http
    builder.AllowAnyHeader();
    builder.AllowAnyMethod();

    //Abilitiamo queste due origini a chiamare api.example.com
    builder.WithOrigins("http://www.example.com", "http://example.com");

    //Se invece vogliamo abilitare ogni origine, usiamo:
    //builder.AllowAnyOrigin();
  });
});
```

[EnableCors] [DisableCors]

```
[EnableCors(policyName: ″AllowOrigins″), Route(″api/[controller]″)]
public class CustomerController : ControllerBase {
  // Le action definite qui useranno la policy Cors ″AllowOrigins″

  // Escludiamo questa action con l´attributo DisableCors
  [HttpPost, DisableCors]
  public void Post([FromBody] string value) {
  }
}
```

CorsMiddleware

```
app.UseCors(″AllowOrigins″);
```

appsettings.json

-
-

User secrets

UserSecretsId

```
<PropertyGroup>
  <TargetFramework>netcoreapp2.1</TargetFramework>
  <UserSecretsId>aspnet-myapp-B3998871-F750…</UserSecretsId>
</PropertyGroup>
```

Configuration Startup

IOptionsMonitor<T>

```
dotnet user-secrets set Smtp:Host smtp.example1.com
```

secrets.json

	%APPDATA%\microsoft\UserSecrets*UserSecretsId*\
	~/.microsoft/usersecrets/*UserSecretsId*/
	~/.microsoft/usersecrets/*UserSecretsId*/

secrets.json

appsettings.json

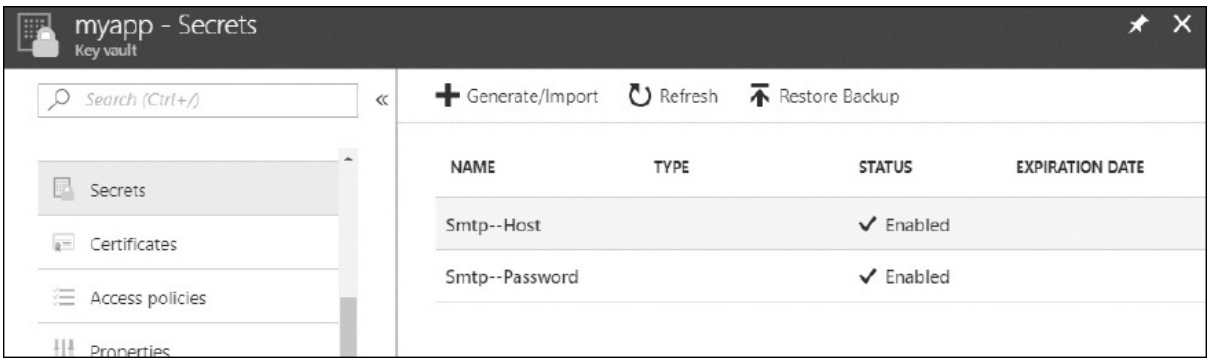
Azure Key Vault

Microsoft.Extensions.Configuration.AzureKeyVault

Program

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
  WebHost.CreateDefaultBuilder(args)
  .ConfigureAppConfiguration((ctx, builder) => {
    var conf = builder.Build();
    var endpoint = conf.GetValue<string>(″VaultEndpoint″);
    var provider = new AzureServiceTokenProvider();
    var callback = provider.KeyVaultTokenCallback;
    var authCallback = new KeyVaultClient.AuthenticationCallback(callback);
    var keyVaultClient = new KeyVaultClient(authCallback);
    var secretManager = new DefaultKeyVaultSecretManager();
    builder.AddAzureKeyVault(endpoint, keyVaultClient, secretManager);
  }).UseStartup<Startup>();
```

appsettings.json



Configuration
Startup
IOptionsMonitor<T>

Usare Data Protection API per cifrare dati

IDataProtectionProvider

```
[HttpPost]
public IActionResult UpdateFiscalCode(string userId, string fiscalCode,
                      [FromServices] IDataProtectionProvider provider)
{
```

```
    //Otteniamo un protector per il purpose ″userData″
    IDataProtector protector = provider.CreateProtector(″userData″);
    string protectedFiscalCode = protector.Protect(fiscalCode);

    //TODO: qui persistiamo il valore di protectedFiscalCode nel db
    SaveFiscalCode(userId, protectedFiscalCode);

    return RedirectToAction(nameof(Index));
  }
```

IDataProtectionProvider
CreateProtector
IDataProtector

IDataProtector

Protect IDataProtector

```
25      [HttpPost]
        0 references
26      public IActionResult UpdateFiscalCode(string userId, string fiscalCode, [FromService
27      {
28      "CfDJ8DoPvoOA_bhCvyy79NA3n7tRyxP_bLE9k264WkZw72vjeqbjYBBHS6rKZgnj47F3m9akqdt-dRK8K8W9azmFrO2fGM
29      aBCzcTkTWiElBwm5gUlVW83PbzCg_jVz0FquypaA"
30          var protectedFiscalCode = protector.Protect(fiscalCode);
31          //...
32          return RedirectToAction(nameof(Index));
33      }
```

Protect
IDataProtector

Unprotect
try…catch

CryptographicException

```
public IActionResult Show(string userId
                        [FromServices] IDataProtectionProvider provider) {
  //Usiamo lo stesso purpose usato all´atto della cifratura
  IDataProtector protector = provider.CreateProtector(˝userData˝);

  //TODO: qui otteniamo la stringa cifrata dal db
  string protectedFiscalCode = GetFiscalCodeFromDb(userId);
  string fiscalCode = null;

  //Decifriamo
  try {
    fiscalCode = protector.Unprotect(protectedFiscalCode);
  } catch (CryptographicException exc) {
    //TODO: logga l´eccezione
  }
  //Passiamo il valore a una view Razor per la visualizzazione
  var data = new UserData { FiscalCode = fiscalCode };
  return View(data);
}
```

Protect Unprotect IDataProtector

Gestire le chiavi crittografiche di Data Protection API

❑

%localappdata%\ASP.NET\DataProtection-Keys

HKLM

DPAPI

- ~/.aspnet/DataProtection-Keys

- %HOME%\ASP.NET\DataProtection-Keys

- ConfigureServices Startup AddDataProtection

ProtectKeysWithDpapi

ProtectKeysWithCertificate

```
//Windows
services.AddDataProtection()
  .PersistKeysToFileSystem(new DirectoryInfo(@"\\SRV1\Share"))
  .ProtectKeysWithDpapi() //Solo per Windows;
//Linux e Mac
var cert = new X509Certificate2("/var/cert.pfx", "password");
services.AddDataProtection()
  .PersistKeysToFileSystem(new DirectoryInfo("/mnt/share"))
  .ProtectKeysWithCertificate(cert); //Disponibile su tutte le piattaforme
```

http://aspit.co/bod

IXmlRepository

```
services.AddDataProtection()
        .SetDefaultKeyLifetime(TimeSpan.FromDays(7));
```

IKeyManager

```
public IActionResult Index([FromServices] IKeyManager keyManager) {
    IReadOnlyCollection<IKey> keys = keyManager.GetAllKeys();
    foreach (var key in keys) {
        key.
    }
```

```
ActivationDate        DateTimeOffset ActivationDate
CreateEncryptor
CreationDate
Descriptor
Equals
ExpirationDate
GetHashCode
GetType
IsRevoked
KeyId
ToString
```

IKeyManager

IKeyManager

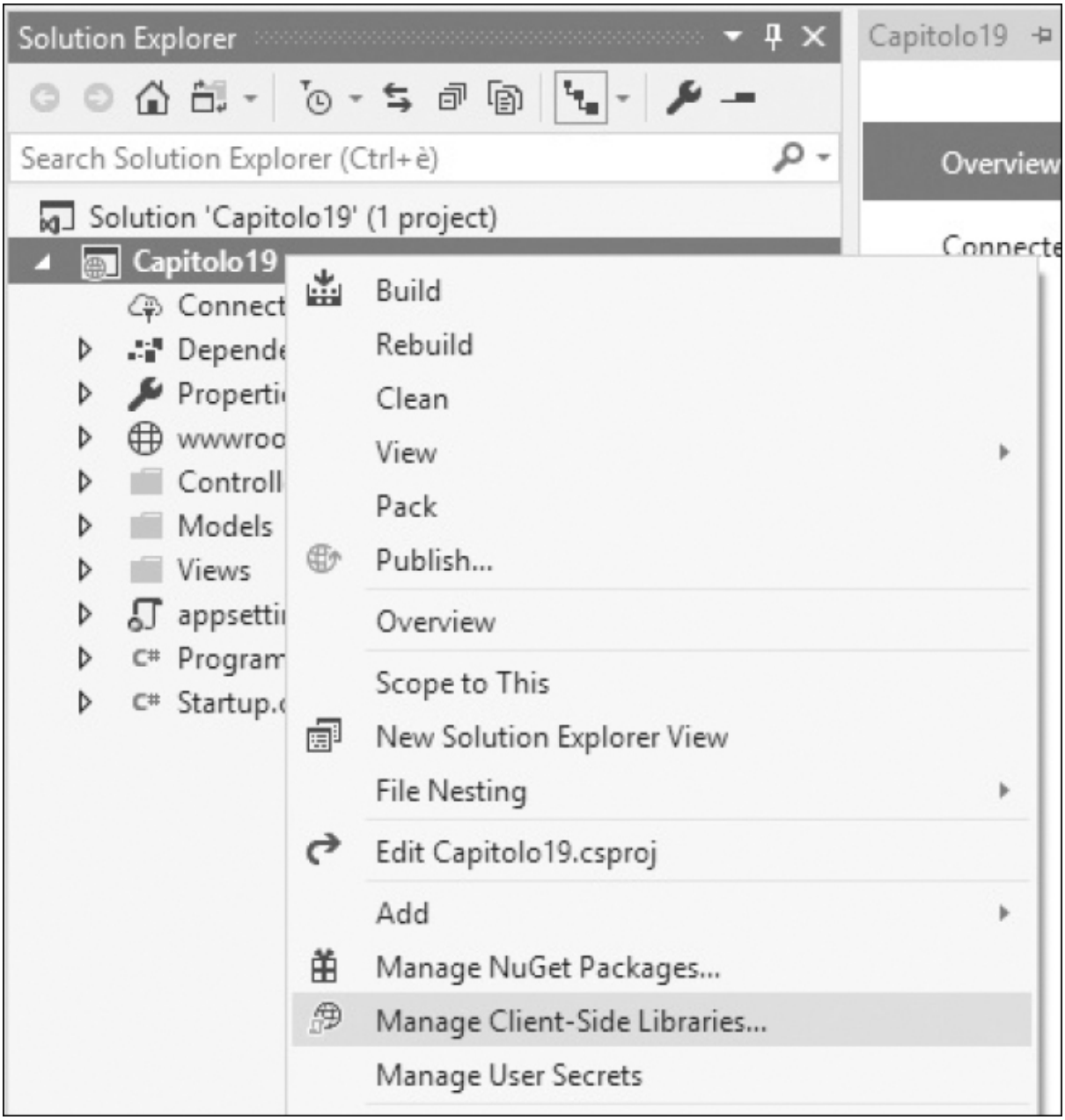
DisableAutomaticKeyGeneration

```
services.AddDataProtection()
         .DisableAutomaticKeyGeneration();
```

IKeyManager

UserManager<TUser>

libman.json


Solution Explorer
Search Solution Explorer (Ctrl+è)
Solution 'Capitolo19' (1 project)
Capitolo19
Connect
Depende
Properti
wwwroo
Controll
Models
Views
appsetti
Program
Startup.c
Capitolo19
Overview
Connecte
Build
Rebuild
Clean
View
Pack
Publish...
Overview
Scope to This
New Solution Explorer View
File Nesting
Edit Capitolo19.csproj
Add
Manage NuGet Packages...
Manage Client-Side Libraries...
Manage User Secrets

defaultProvider

libman.json

version

1.0

defaultDestination

libraries

libman.json Libraries

library provider destination files

library

nome@versione

provider

defaulProvider provider

provider

defaulProvider destination

defaultDestination

files

files

```
{
  ″version″: ″1.0″,
  ″defaultProvider″: ″cdnjs″,
  ″libraries″:
  [
    {
      ″library″: ″jquery@3.3.1″,
      ″destination″: ″wwwroot/lib/jquery″,
      ″files″: [″jquery.slim.min.js″ ]
    },
    {
      ″library″: ″twitter-bootstrap@4.1.1″,
      ″destination″: ″wwwroot/lib/bootstrap/″
    },
    {
      ″library″: ″vue@2.5.16″,
      ″destination″: ″wwwroot/lib/vue/″
    }
  ]
}
```

defaultProvider

library

@

files

Maggiori informazioni su jQuery sono disponibili all'url: http://aspit.co/a10.

Effettuare ricerche nel DOM

querySelectorAll

document

querySelectorAll

$

```
//ricerca un elemento per id
const objects = $(´#textboxId´);

//ricerca un elemento per id fornito da ASP.NET
const objects = $(´#@Html.IdFor(m => m.Name)´);

//ricerca elementi per classe
const objects = $(´.classe´);

//ricerca tag p all´interno dei div con classe ˝contenitore˝
const objects = $(´div.contenitore p´);
```

objects

Manipolare gli oggetti

addClass

```
const objects = $(´.myClass´)
  .addClass(´selected´)
  .addClass(´anotherClass´);
```

addClass

selected

addClass

addClass

addClass

append	
html	innerHTML innerHTML
text	html innerText
after before	

empty	
remove	
attr	
removeAttr	
val	value value
addClass removeClass toggleClass	
hide show toggle	

Gestire gli eventi

on

off

on on off

```
//si sottoscrive all´evento click di un pulsante
$(´#myButton´).on(´click´, handler);

//si cancella dall´evento click di un pulsante
$(´#myButton´).off(´click´, handler);

function handler() {
  alert(´bottone cliccato´);
}
```

trigger

$

Effettuare chiamate AJAX

XMLHttpRequest

ajax

ajax

cache	false
data	
headers	
jsonp	
timeout	
type	
url	
contentType	application/x-www-form-urlencoded; charset=UTF-8 applicaton/json

ajax

Deferred

- done
- fail

- always

ajax.

```
$.ajax({
  url: ´/userhandler´
  data: { userId: 1 }
})
.done(function(result){
  alert(result);
})
.fail(function(){
  alert(´errore´);
})
.always(function(){
  alert(´chiamata terminata´);
});
```

ajax

ajax

get getJSON post load

get

getJSON

post

load

```
$.get(´/userhandlerget´, { userId: 1 });
$.getJSON(´/userhandlerjson´, { userId: 1 });
$.post(´/userhandlerpost´, { userId: 1 });
$(´#divid´).load(´/userhandlerload´, { userId: 1 });
```

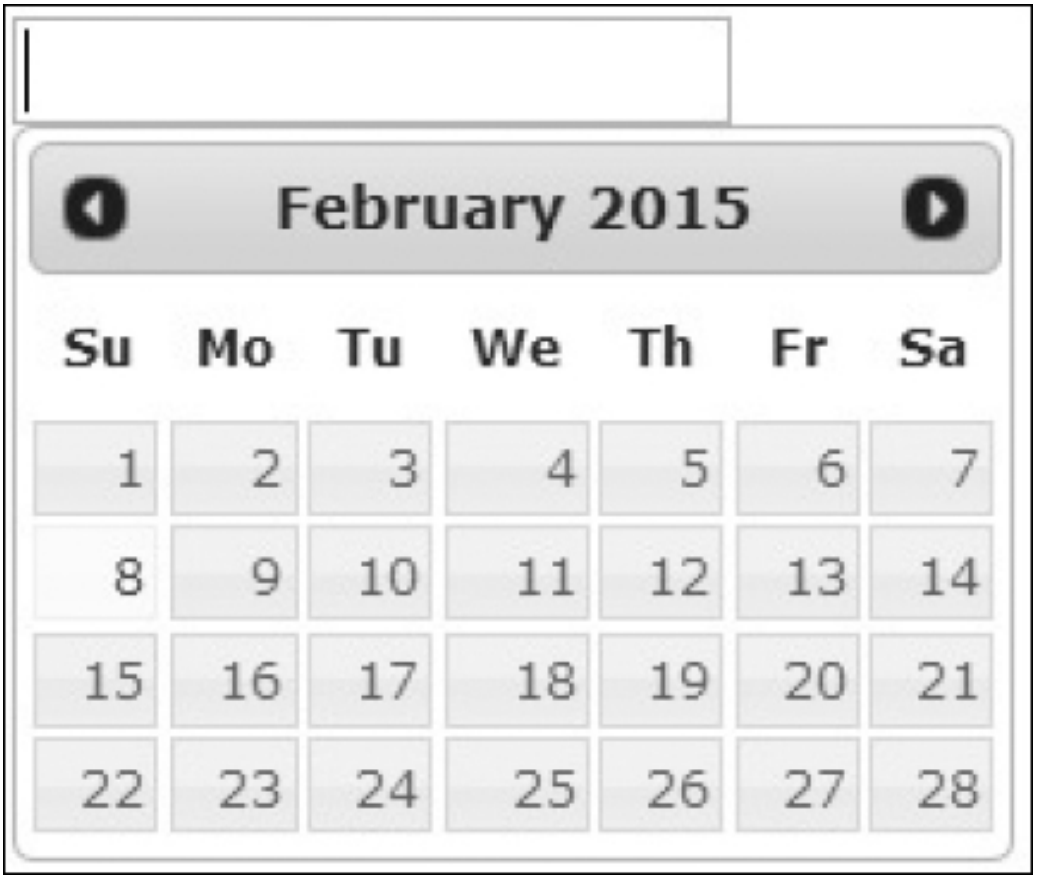
jQueryUI

- accordion

- autocomplete
- datepicker
- dialog
- tabs

```
//html
<input type=″text″ id=″datepicker″>
//JavaScript
$(function() {
  $(´#datepicker´).datepicker();
});
```

datepicker



http://aspit.co/ain

jQuery mette a disposizione anche delle animazioni che scattano quando si mostra o nasconde un oggetto o quando si aggiunge o rimuove una classe CSS. Inoltre, mette a disposizione anche dei

comportamenti per arricchire la nostra applicazione, come il drag&drop, l'ordinamento di oggetti e così via.

Al momento della scrittura di questo libro, Bootstrap è giunto alla versione 4.1.1 ed è disponile all'indirizzo http://aspit.co/a1y.

- `Alert`
- `Carousel`

- Collapse
- Modal
- Navbar
- Tooltip
- Tab

Il JavaScript di Bootstrap è incluso in un solo file, ma ha una dipendenza da una libreria di terze parti chiamata popper. Questa non è distribuita da Bootstrap e va quindi inclusa negli script della pagina.

```
<div class="alert alert-success alert-dismissible" id="message">
  <strong>Terminato!</strong> Hai completato la registrazione con successo.
  <button type="button" class="close" data-dismiss="alert">
    <span>&times;</span>
  </button>
</div>
```

alert alert-success alert-dismissible

data-dismiss alert

data-dismiss alert

close alert

close.bs.alert closed.bs.alert

```
$(´#message´).alert(´close´); //chiude l´alert
$(´#message).on(´close.bs.alert´, function () {
  console.log(´In chiusura´);
});
$(´#message).on(´closed.bs.alert´, function () {
  console.log(´chiuso´);
});
```

modal

Il sito di Vue è disponibile a questo indirizzo: http://aspit.co/bnd.

`Vue` `el`

`data`

```
<div id="binding" style="border: 1px solid black">
  <span>{{ message }}</span>
</div>
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
<script>
  const app = new Vue({
    el: '#binding',
    data: {
      message: 'messaggio via binding'
    }
  });
</script>
```

messaggio via binding

message

v-model

```
<input type="text" v-model="name" />
{{name}}
<script>
  const app = new Vue({
    el: '#binding',
    data: {
      name: 'Stefano Mostarda'
    }
  });
</script>
```

name

v-bind

```
<div id="binding">
  <input type="checkbox" v-model="selected" />check
  <button v-bind:disabled="!selected">button</button>
  <div v-bind:class="{ 'd-none': !selected }">Contenuto</div>
</div>

<script>
  const app = new Vue({
    el: '#binding',
    data: {
      selected: true
    }
  });
</script>
```

v-bind

disabled selected

false

class

d-none

v-if

false

v-for

```
<div id="binding">
  <ul>
    <li v-for="item in people">
      {{ item.firstName }} {{ item.lastName }}
    </li>
      </ul>
</div>

<script>
  const app = new Vue({
    el: '#binding',
    data: {
      people: [
        { firstName: 'Stefano', lastName: 'Mostarda' },
        { firstName: 'Daniele', lastName: 'Bochicchio' }
      ]
    }
  });
</script>
```

li

people

v-on

methods

```
<div id="binding">
  <button v-on:click="showMessage">show message</button>
  <span v-if="show">Messaggio visibile</span>
</div>

<script>
  const app = new Vue({
    el: '#binding',
    data: {
      show: false
    },
    methods: {
      showMessage: function () {
        this.show = true;
      }
    }
  });
</script>
```

v-on

:

this

value

```
<div id="binding">
  <input type="text" asp-for="Name" v-model="name" />
  {{name}}
</div>

<script>
  const app = new Vue({
    el: '#binding',
    data: {
      name: '@Model.Name'
    }
  });
</script>
```

libman.json

20

Gli strumenti di Visual Studio per lo sviluppo web

Nel precedente capitolo abbiamo introdotto alcune librerie JavaScript che migliorano l'esperienza di sviluppo di applicazioni web ibride fornendo componenti già pronti per l'uso che sfruttano sia HTML che JavaScript. In questo capitolo vediamo come sfruttare alcuni strumenti per ottimizzarne le performance e per pre-processare alcuni file.

Quando scriviamo codice, prima che questo venga eseguito, la macchina deve compilarlo in codice nativo. Quando sviluppiamo per il Web, possiamo traslare questa affermazione dicendo che quando creiamo file CSS o JavaScript, dobbiamo precompilarli prima di distribuirli via web. Esistono diversi casi in cui quanto appena detto corrisponde a realtà.

Quando referenziamo un file JavaScript all'interno di una pagina web, questo viene inviato al client così com'è. Se una pagina referenzia più file JavaScript il client esegue una richiesta per ogni file. Queste affermazioni portano alla considerazione che avere file molto grandi o referenziare troppi file può causare problemi di performance di rete della nostra applicazione. Questa stessa affermazione si applica non solo ai file JavaScript, ma anche ai file CSS. Nel corso degli anni si è ricorso a diverse tecniche per mitigare il problema e si è arrivati a due soluzioni complementari: minification, cioè la capacità di ridurre le dimensioni di un file, e bundling, cioè la capacità di unire più file in un solo file. Entrambe le tecniche prevedono che i file JavaScript e CSS subiscano una sorta di precompilazione prima di essere usate.

Per fare un altro esempio, nei file CSS molto spesso ci sono tante informazioni ripetute come i colori, le dimensioni, i margini e così via. Quando dobbiamo modificare uno di questi valori, per esempio un colore, dobbiamo scorrere i file CSS e modificare il valore ovunque. È chiaro che un'organizzazione del genere comporta grosse difficoltà di manutenzione con alte probabilità di errore. Per questo motivo sono state inventate sintassi simil-CSS (come SCSS) che vengono pre-processate e che producono in output file CSS da inviare ai client.

Questi esempi rendono chiaro il perché non sia sempre una buona idea sviluppare e inviare direttamente il nostro sorgente al client, ma sia molto meglio preprocessarlo per avere un ventaglio di opportunità molto più ampio.

Nel corso del capitolo parleremo delle tecniche appena menzionate e di come metterle in pratica sfruttando alcuni strumenti di terze parti integrati in Visual Studio (Gulp, Grunt, e NPM).

Bundling e minification

Queste due tecniche permettono di ottimizzare in modo importante le prestazioni di rete della nostra applicazione. Molte applicazioni di tuning delle applicazioni web considerano la mancanza di bundling e minification un problema e lo segnalano.

> *Due esempi di applicazioni di tuning dei siti web sono PageSpeed, sviluppato da Google e disponibile all'indirizzo http://aspit.co/bph, e YSlow disponibile all'indirizzo http://aspit.co/bpj.*

Inoltre, i motori di ricerca penalizzano siti (soprattutto mobile) che non sfruttano queste tecniche. Il risultato è che il rank di questi siti è più basso e quindi appaiono più in basso nelle ricerche.

Minification

La minification è la tecnica che prende in input un file e ne riduce le dimensioni applicando una serie di regole. Alcune di queste regole

prevedono la modifica del nome, al fine di renderlo più corto, delle variabili interne e dei parametri dei metodi, la rimozione di commenti, spazi e altri caratteri inutili e altro ancora. Il risultato è un file poco comprensibile ma di dimensioni notevolmente ridotte. La percentuale di riduzione del codice dipende molto dai nomi originali, ma si stima che si possano ridurre le dimensioni originali fino al 60%. Prendiamo come esempio il seguente codice JavaScript che mostra il codice originale e il codice minificato.

Esempio 20.1

```
//codice originale
var calculateArea = function(xSize, ySize) {
  //multiply input parameters
  return xSize * ySize;
}

//codice minificato
var calculateArea=function(a,b){return a*b};
```

Il codice originale è di 103 caratteri mentre quello minificato ne conta 44 con un'ottimizzazione del 55%. In un file di così piccole dimensioni la differenza è risibile, ma al crescere delle dimensioni del file il guadagno è notevole. Grazie a questa tecnica, sulla rete viaggiano molti meno dati e quindi la nostra applicazione è molto più veloce da caricare.

Lo stesso discorso si può applicare non solo al codice JavaScript ma anche ai file CSS, come nel prossimo esempio.

Esempio 20.2

```
//codice originale
.myStyle {
  border-width: 1px;
  border-style: solid;
  border-color: black;
  background-color: red;
}

.otherstyle{
  color: #000;
  white-space: nowrap;
  font-size: 10px
}

//codice minificato
```

```
.myStyle{border-width:1px;border-style:solid;border-color:#000;
background-color:red}.otherstyle{color:#000;white-space:nowrap;
font-size:10px}
```

In questo caso, il codice originale è di 109 caratteri mentre quello minificato è di 85 caratteri, con un risparmio di circa il 25%. Anche in questo caso, con file di così piccole dimensioni, il vantaggio è minimo ma, all'aumentare delle dimensioni del file originale, il risparmio diventa più consistente e si hanno maggiori benefici. Se sommiamo il risparmio che possiamo ottenere minificando sia CSS sia JavaScript, appare evidente che il miglioramento delle performance di rete è veramente importante. Passiamo ora a vedere il bundling, che offre ulteriori vantaggi.

Bundling

Il bundling è la tecnica che prevede l'unione di più file in un unico file, al fine di ridurre le richieste di rete che il client fa verso il server. Se questi file sono anche minificati, otteniamo un unico file minificato e abbiamo quindi limitato al massimo i dati da scaricare.

Creare un solo unico file JavaScript e un solo unico file CSS può non essere la soluzione migliore. Se da un lato riduciamo al minimo il numero di file scaricati e la loro dimensione, dall'altro non sfruttiamo le capacità di parallelismo dei browser. Infatti, i browser sono in grado di scaricare più file contemporaneamente (il numero di file contemporanei varia da browser a browser) e questo significa che avere pochi file di dimensioni ridotte potrebbe essere più performante di avere un solo file di grandi dimensioni. Come sempre, occorre fare dei test prima di scegliere una strada o l'altra.

È chiaro che l'insieme di bundling e minification rappresenta una scelta praticamente obbligata per creare applicazioni che siano performanti dal punto di vista della rete. Tuttavia, mentre stiamo sviluppando quest'ottimizzazione, è inutile, anzi addirittura dannosa nei casi in cui dobbiamo eseguire il debug dei file JavaScript. In questi casi può venire in aiuto ASP.NET Core.

Integrazione con ASP.NET Core

Nel Capitolo 6 abbiamo introdotto il tag helper `environment`, che permette di renderizzare frammenti di HTML in base all'ambiente in cui ci troviamo. Nel nostro caso usiamo questo tag helper per renderizzare dei tag `link` o `script` differenti a seconda dell'ambiente. Nel caso in cui ci troviamo nell'ambiente di sviluppo, utilizziamo i codici sorgenti; se, invece, ci troviamo nell'ambiente di produzione, utilizziamo i file sottoposti a bundling e minification come viene mostrato nell'esempio.

Esempio 20.3

```
<environment include="Development">
  <link rel="stylesheet" href="~/css/bootstrap.css" />
  <link rel="stylesheet" href="~/css/site.css" />
</environment>
<environment exclude="Development">
  <link rel="stylesheet" href="~/dist/bundle.min.css" />
</environment>
```

In questo esempio abbiamo utilizzato il tg `link` per referenziare file CSS ma, allo stesso modo, possiamo usare il tag `script` per referenziare file JavaScript. Entrambi i tag sono in realtà anche dei tag helper, il cui utilizzo è spiegato nella prossima sezione.

Link e Script tag helper

Nell'Esempio 20.3 abbiamo visto che il tag `link` nell'ambiente di sviluppo punta al file di Bootstrap e a un altro file CSS, mentre negli altri ambienti punta a un unico file che contiene entrambi i file in bundling. Tuttavia, in un'applicazione reale, la referenza al file di Bootstrap non dovrebbe puntare a un file locale, bensì a un file servito da una CDN, così da ottimizzare al massimo le performance, in quanto molto probabilmente il file si trova già nella cache del browser.

Essendo il file fornito da una CDN su cui non abbiamo controllo, dobbiamo anche prevedere un meccanismo di fallback che fornisca il file dal server locale qualora la CDN non dovesse essere in grado di fornirlo. In questi casi entra in gioco il tag helper `link`, che offre una serie di

proprietà che eseguono un test per capire se il file è stato scaricato dalla CDN e specificano da quale sorgente alternativa scaricare il file nel caso il test fallisca. Le proprietà sono:

- `asp-fallback-test-class`: specifica una classe CSS che si trova nel file che intendiamo scaricare dlla CDN;
- `asp-fallback-test-property`: specifica una proprietà della classe CSS che si deve trovare nella classe CSS specificata in precedenza;
- `asp-fallback-test-value`: specifica il valore che deve avere la proprietà specificata in precedenza;
- `asp-fallback-href`: specifica l'url del file da scaricare nel caso il test fallisca.

L'Esempio 20.4 mostra come scaricare il file CSS di Bootstrap da una CDN, come specificare il test (verificando che esista una classe CSS `sr-only` con la proprietà `position` impostata su `absolute`) e come fornire un fallback dal server locale.

Esempio 20.4

```
<link rel="stylesheet" href="
  https://ajax.aspnetcdn.com/ajax/bootstrap/4.1.1/css/bootstrap.min.css"
  asp-fallback-href="~/css/bootstrap.min.css"
  asp-fallback-test-class="sr-only" asp-fallback-test-property="position"
  asp-fallback-test-value="absolute" />
```

Il tag helper `link` espone anche la proprietà booleana `asp-append-version` tramite la quale indichiamo ad ASP.NET Core di includere un hash del file nell'url, nel caso il file sia servito da locale. Questa funzionalità torna utile quando dobbiamo servire un file da locale e cambiamo il contenuto del file. Se il file si trova già nella cache del browser e non includiamo l'hash, il browser utilizzerà la versione in cache e ignorerà la nuova versione. Se invece mettiamo l'hash nell'url, il

server invierà al client un url diverso da quello precedente e quindi il browser sarà forzato a eseguire il download del nuovo file.

Il tag `script` non è molto diverso dal tag `link` in quanto svolge gli stessi compiti, ma con una differente sintassi dovuta al fatto che il test per capire se il file JavaScript è stato scaricato dalla CDN non è come il test per i CSS. Invece delle tre proprietà di test per i CSS, abbiamo solamente la proprietà `asp-fallback-test`, che contiene un'espressione JavaScript che deve essere vera per confermare il download del file dalla CDN. Nel caso l'espressione sia falsa, si procede al download dal fallback espresso dalla proprietà `asp-fallback-href`.

Esempio 20.5

```
<script
  src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/bootstrap.min.js"
  asp-fallback-src="~/scripts/bootstrap.min.js"
  asp-fallback-test="window.jQuery.fn.modal">
</script>
```

L'integrazione di bundling, minification e CDN con ASP.NET Core è semplice grazie ai tag `environment`, in combinazione con `script` e `link`. Tuttavia non abbiamo ancora visto come applicare minification e bundling ai file sorgenti all'interno di Visual Studio. Questi task possono essere eseguiti attraverso due strumenti, Gulp e Grunt, che hanno in comune il fatto di dover essere scaricati da NPM, che è l'argomento della prossima sezione.

Utilizzare NPM

NPM è l'acronimo di NodeJS Package Manager ed è un repository di package JavaScript oramai diffusissimo tra tutti gli sviluppatori client. All'interno di NPM ci sono package di librerie JavaScript così come anche strumenti (detti anche plugin) di sviluppo, come Gulp e Grunt.

> *Microsoft consiglia di utilizzare NPM esclusivamente per scaricare plugin di sviluppo e non librerie (in quanto queste possono essere scaricate tramite LibMan). Tuttavia questa è*

solo una raccomandazione e non ci sono problemi a scaricare anche librerie da NPM.

NPM si abilita all'interno del progetto creando il file `package.json` e salvandolo nella root del progetto. Visual Studio semplifica questo compito, offrendo un template nella finestra per creare un nuovo file come mostra la Figura 20.1.

Il file creato da Visual Studio prevede le proprietà minime per il funzionamento di NPM (`name`, `version`, `private`), più la proprietà `devDependencies` che è la più importante per i nostri scopi. Questa proprietà, inizialmente vuota, contiene il dictionary all'interno del quale specifichiamo i plugin che vogliamo scaricare: la chiave rappresenta il nome del plugin, mentre il valore rappresenta la versione. Ogni volta che salviamo il file, Visual Studio si connette a NPM e scarica i componenti lanciando il comando `npm install`.

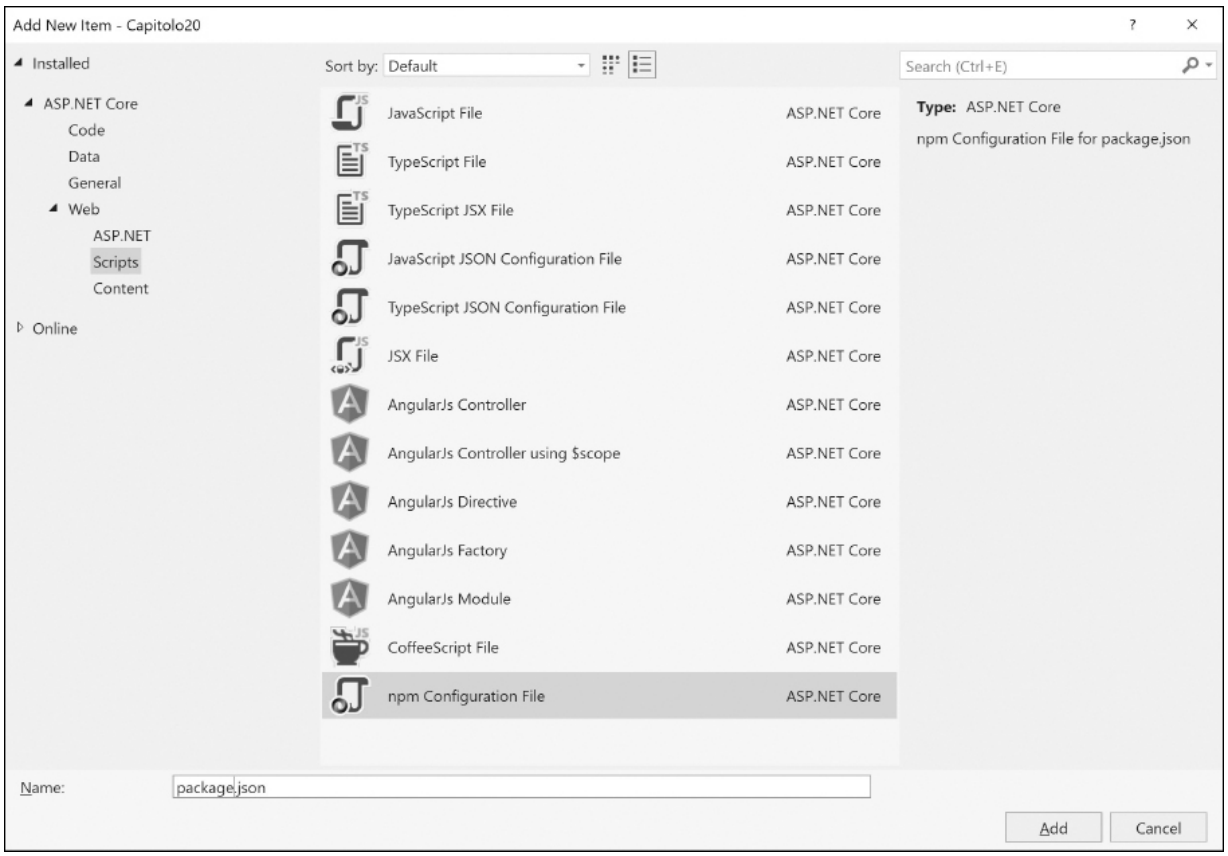


Figura 20.1 – La finestra di Visual Studio che permette di creare il file `package.json`.

La versione può essere espressa in tre modi:

- numero di versione: scarica l'esatta versione;
- numero di versione preceduto da ^: scarica l'ultima major version;
- numero di versione preceduto da ~: scarica l'ultima minor version;

L'Esempio 20.6 mostra il file `package.json` con le diverse opzioni per specificare la versione.

Esempio 20.6

```
{
  "version": "1.0.0",
  "name": "asp.net",
  "private": true,
    "devDependencies": {
      "grunt": "^1.0.2",
      "gulp": "~3.9.1",
      "gulp-minify": "2.1.0",
    }
}
```

Oltre alla proprietà `devDependencies` esiste la proprietà `dependencies`, che viene usata per scaricare librerie da usare nel codice, invece che plugin usati a design time, e che ha la stessa struttura di `devDependencies` all'interno del file `package.json`. Il motivo per cui esistono due sezioni è per organizzare al meglio il file `package.json` e per facilitare il deploy in produzione, in quanto lanciando il comando `npm install --only=production`, si scaricano solo le librerie referenziate nella sezione `dependencies`.

NPM è uno strumento estremamente potente che tramite linea di comando offre molte altre opzioni. Tuttavia non dobbiamo preoccuparci di conoscere il comando e le sue opzioni, in quanto l'integrazione con

Visual Studio esegue tutto al posto nostro nel momento in cui salviamo il file.

> *Per maggiori informazioni su NPM è disponibile la documentazione sul sito ufficiale all'indirizzo http://aspit.co/bov.*

Ora che sappiamo come recuperare i plugin di sviluppo necessari a processare i nostri file JavaScript e CSS, vediamo come utilizzarli partendo da Gulp.

Utilizzare Gulp

Gulp è un motore (detto task runner) che esegue una dietro l'altra funzioni JavaScript (dette task) che hanno un input e ritornano un output che andrà in input alla funzione successiva. Gulp non è a conoscenza di cosa facciano i task e di quale sia il loro input e il loro output, in quanto queste specifiche appartengono ai task. Gulp è solo responsabile dell'invocazione dei task e del trasporto dei dati che questi ricevono e ritornano.

> *Microsoft non è proprietaria di Gulp: questo strumento è open source su github ed è stato solamente integrato in Visual Studio.*

La definizione della catena di task è fatta nel file `gulpfile.js` situato nella root del progetto. In testa al file importiamo le dipendenze del nostro script e, successivamente, specifichiamo i vari task da eseguire.

Per importare una dipendenza, la dobbiamo scaricare da NPM e utilizzare il metodo `require` con il nome della dipendenza. Visto che il nostro scopo è quello di eseguire bundling e minification, dobbiamo scaricare i plugin che eseguono queste operazioni tramite NPM e referenziarli nel file `gulpfile.json` per orchestrarli con Gulp. I plugin sono `gulp-concat` (che concatena più file), `gulp-minify` (che minifica i file JavaScript) e `gulp-clean-css` (che minifica i file CSS). L'Esempio 20.7 mostra come importare queste dipendenze.

Esempio 20.7

```
const gulp = require(´gulp´);
const concat = require(´gulp-concat´);
const cleancss = require(´gulp-clean-css´);
const minify = require(´gulp-minify´);
```

Per creare un task, utilizziamo il metodo `task` della classe `gulp`, alla quale passiamo il nome del task e una funzione all'interno della quale specifichiamo gli step necessari a completare il task. Ogni step prende in input l'output dello step precedente, detto stream, e ritorna a sua volta uno stream che viene passato allo step successivo. Arrivati all'ultimo step, il task si conclude.

Generalmente, il primo step è il metodo `src` della classe `gulp`, al quale passiamo in input un array di file. Questi file vengono inseriti in uno stream che passiamo allo step successivo, invocato tramite il metodo `pipe`, sempre della classe `gulp`.

`pipe` accetta in input il metodo che lavora lo stream proveniente dallo step precedente e che ritorna un nuovo stream. Questo viene preso da `pipe` che lo passa allo step successivo invocato di nuovo tramite `pipe`, creando così un processo che si itera fino all'ultima chiamata.

Nell'Esempio 20.8 creiamo il task `bundle-js` per lavorare i file JavaScript. In questo esempio sfruttiamo il metodo `src` per recuperare i file JavaScript da lavorare, il metodo `minify` per eseguirne la minification, il metodo `concat` per eseguire il bundling (specificando il nome del file di bundle) e, infine, il metodo `dest` della classe `gulp` per salvare il file nel percorso specificato.

Nello stesso esempio creiamo anche il task `bundle-css`, per lavorare i file CSS, che sfrutta lo stesso pattern visto in precedenza: selezioniamo i file con il metodo `src`, ne eseguiamo la minification con il metodo `cleancss`, ne eseguiamo il bundling con il metodo `concat` e, infine, salviamo il file con il metodo `dest`.

Esempio 20.8

```
gulp.task(´bundle-js´, function () {
```

```
  return gulp
    .src([´./wwwroot/js/site.js´, ´./wwwroot/js/site2.js´])
    .pipe(minify())
    .pipe(concat(´bundle.min.js´))
    .pipe(gulp.dest(´./wwwroot/dist´));
});

gulp.task(´bundle-css´, function () {
  return gulp
    .src([´./wwwroot/css/site.css´, ´./wwwroot/css/site2.css´])
    .pipe(cleancss())
    .pipe(concat(´bundle.min.css´))
    .pipe(gulp.dest(´./wwwroot/dist´));
});
```

Una volta salvato il file, dobbiamo eseguirne i task tramite la CLI di Gulp. Tuttavia l'integrazione di Gulp all'interno di Visual Studio permette di lanciare i task sfruttando una UI (visibile nella Figura 20.2) accessibile tramite il menù View - Other Windows - Task Runner Explorer.

La maschera mostra sulla sinistra i task nel file `gulpfile.js`. Con un doppio click del mouse su uno dei task, eseguiamo il task e, sulla destra, la finestra ne mostra il risultato. Sulla destra c'è anche il tab `bindings`, tramite il quale possiamo impostare l'esecuzione di un task prima o dopo la compilazione, all'apertura del progetto o nella fase di clean del progetto.

> *Per maggiori informazioni su Gulp, è disponibile la documentazione sul sito ufficiale, all'indirizzo: http://aspit.co/bow.*

Come abbiamo detto, Gulp non è il solo task runner in Visual Studio, in quanto c'è anche Grunt, che tratteremo nella prossima sezione.

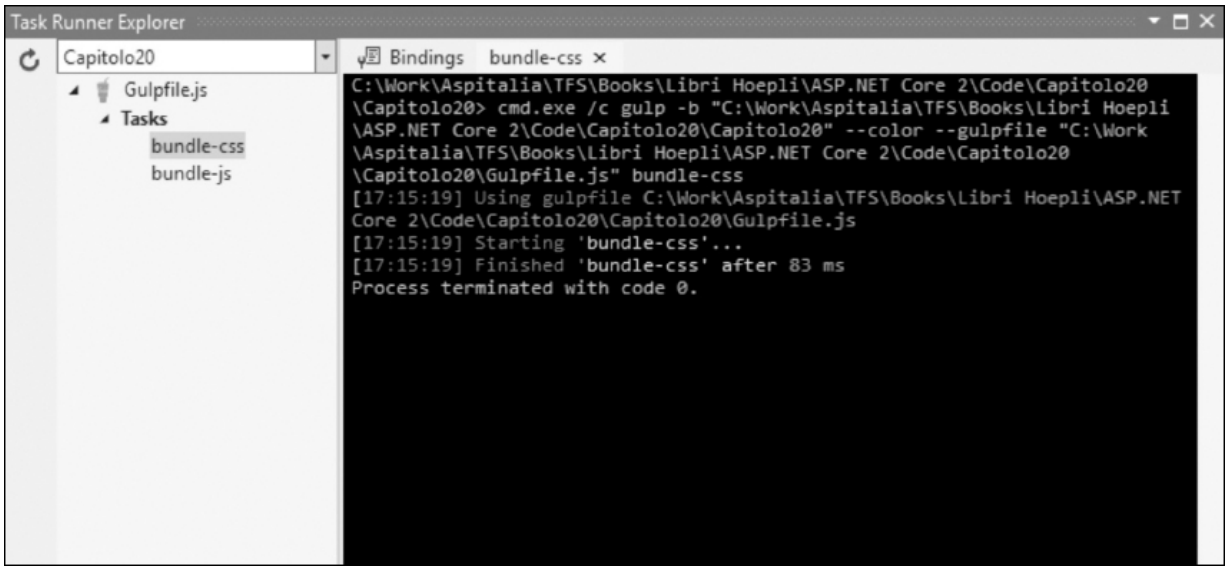


Figura 20.2 – La finestra di Visual Studio che permette di lanciare i task di Gulp.

Utilizzare Grunt

Grunt è un altro task runner integrato in Visual Studio, che svolge le stesse funzioni di Gulp ma in modo leggermente diverso e sfruttando altri plugin. La scelta tra Gulp e Grunt è dettata dalle conoscenze e dai gusti personali ma c'è da sottolineare che mentre Gulp è costantemente manutenuto ed evoluto, Grunt dopo la release 1.0 ha avuto solo un paio di minor release per bug fixing.

Prima di passare a vedere come funziona Grunt, dobbiamo specificare che i plugin di Grunt e Gulp non sono compatibili e quindi, per poter eseguire minification e bundling, dobbiamo scaricarne di nuovi da NPM. Questi nuovi plugin sono `Grunt` per scaricare Grunt, `grunt-contrib-uglify` per lavorare i file JavaScript e `grunt-contrib-cssmin` per lavorare i file CSS.

Per abilitare Grunt in Visual Studio, dobbiamo creare il file `gruntfile.js` nella root del progetto e modificarlo per aggiungere i task da eseguire. Poiché Grunt viene eseguito da NodeJS, il file deve dichiarare innanzitutto l'entry point, impostando la variabile `module.exports` con una funzione all'interno della quale impostiamo i task da eseguire.

Esempio 20.9

```
module.exports = function (grunt) {
  // Configurazione grunt
}
```

La funzione che agisce come entry point accetta in input un'istanza della classe tramite la quale configuriamo Grunt. Il primo metodo che utilizziamo è `initConfig`, al quale dobbiamo passare un oggetto JSON che contiene i dati di configurazione dei plugin e anche altro. La prima proprietà dell'oggetto JSON è `pkg`, che facciamo puntare al file `package.json`, così che Grunt possa assicurarsi di usare la versione corretta dei plugin, scaricandoli automaticamente tramite NPM, se necessario. Successivamente creiamo una proprietà per ogni plugin che utilizziamo, dove il nome della proprietà deve corrispondere al nome del plugin e il valore assegnato alla proprietà è un oggetto che configura il plugin. Quest'oggetto è specifico per ogni plugin quindi dobbiamo consultarne la documentazione per capire come configurarlo. Ogni singolo plugin configurato corrisponde a un task che possiamo invocare.

Nel prossimo esempio vediamo come creare l'oggetto JSON per eseguire bundling e minification di JavaScript e CSS attraverso i plugin specificati in precedenza.

Esempio 20.10

```
grunt.initConfig({
  pkg: grunt.file.readJSON(´package.json´),
  uglify: {
    build: {
      files: {
        ´./wwwroot/dist/bundle.min.js´: [
          ´./wwwroot/js/site.js´,
            ´./wwwroot/js/site2.js´]
      }
    }
  },
  cssmin: {
    build: {
      files: {
        ´wwwroot/dist/bundle.min.css´: [
          ´wwwroot/css/site.css´,
            ´wwwroot/css/site2.css´]
      }
```

```
    }
  }
});
```

Il plugin `grunt-contrib-uglify` ha come nome `uglify` e l'oggetto in input specifica i file JavaScript da processare e il nome del file di bundle. Il plugin `grunt-contrib-cssmin` ha come nome `cssmin` e la medesima configurazione di `uglify` con la sola differenza di processare i file CSS. Una volta configurati i plugin, dobbiamo utilizzare il metodo `loadNpmTask` dell'istanza di Grunt per importare i plugin. L'utilizzo di questo metodo è mostrato nell'Esempio 20.11.

Esempio 20.11

```
grunt.loadNpmTasks(´grunt-contrib-uglify´);
grunt.loadNpmTasks(´grunt-contrib-cssmin´);
```

L'ultimo metodo dell'oggetto di Grunt da invocare è `registerTask`, che crea task combinando quelli definiti nel JSON passato al metodo `initConfig`.

`registerTask` prende in input il nome del task e un array di stringhe che corrispondono ai nomi dei plugin. Un utilizzo è mostrato nel prossimo esempio.

Esempio 20.12

```
grunt.registerTask(´build´, [´uglify´, ´cssmin´]);
```

L'ultimo step consiste nel lanciare i task da Visual Studio attraverso la finestra Task Runner Explorer, esattamente come abbiamo visto per Gulp. L'unica variante consiste nel fatto che la finestra mostra un task per ogni plugin configurato più i task configurati con il metodo `registerTask` sotto il nodo Alias Task come mostra la Figura 20.3.

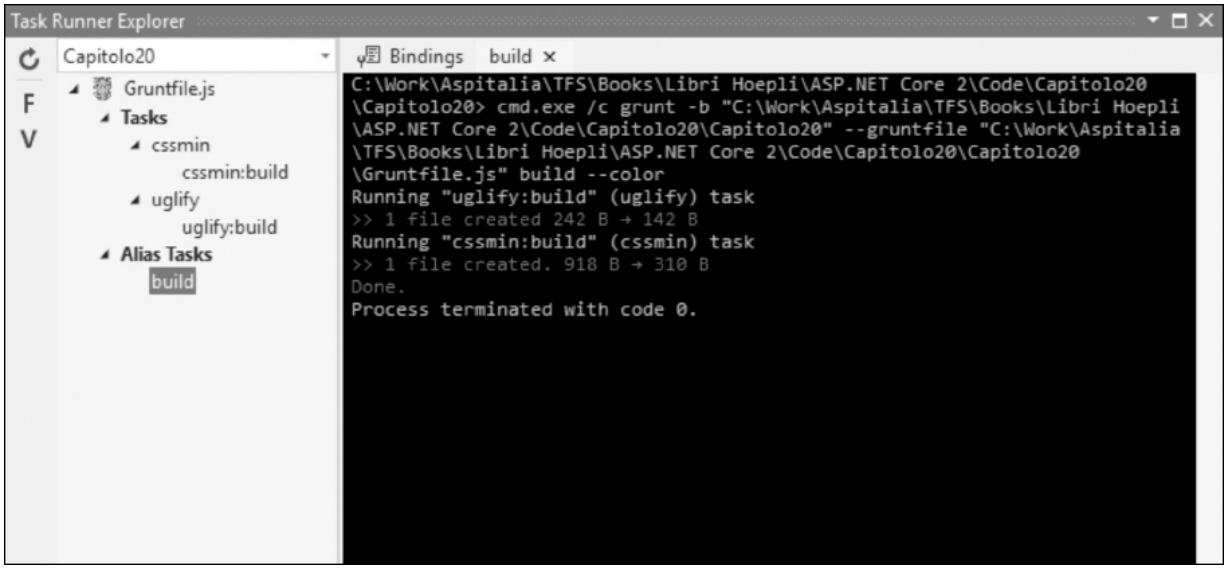


Figura 20.3 – La finestra di Visual Studio che permette di lanciare i task di Grunt.

Per maggiori informazioni su Grunt, è disponibile la documentazione sul sito ufficiale, all'indirizzo: http://aspit.co/box.

Conclusioni

In questo capitolo abbiamo fatto la conoscenza di due tecniche molto importanti per le performance di rete di un'applicazione: minification e bundling. Oltre a queste tecniche, possiamo anche abilitare la compressione GZip tramite un middleware o sul reverse proxy, con lo scopo di ottimizzare ulteriormente il traffico dati.

Inoltre, abbiamo visto come Visual Studio non offra solo strumenti dedicati a chi sviluppa funzionalità lato server ma anche funzionalità per gli sviluppatori che lavorano prettamente sulla UI e che necessitano maggiormente di queste funzionalità.

Grazie all'integrazione di Visual Studio con strumenti come NPM, Gulp, Grunt e, implicitamente, NodeJS, possiamo pre-processare qualunque tipo di file prima di utilizzarlo nelle nostre pagine e risparmiare così lavoro manuale a design time, ottimizzando le performance a run time.

Ora cambiamo argomento e parliamo di come creare Single Page Application in Visual Studio con ASP.NET Core e Angular.

21

Single Page Application e ASP.NET Core

Negli ultimi due capitoli abbiamo introdotto i principali framework, librerie e strumenti per lo sviluppo lato client di applicazioni ibride. Abbiamo visto, inoltre, come Visual Studio permetta facilmente l'integrazione di questi strumenti all'interno delle nostre applicazioni, così da rendere lo sviluppo più semplice rispetto al passato.

In questo capitolo vediamo come Visual Studio permetta lo sviluppo non solo di applicazioni ibride ma anche di Single Page Application (SPA d'ora in poi). Le SPA sono applicazioni che girano interamente sul browser e che sfruttano il server solamente per recuperare i file JavaScript, HTML e CSS e per leggere e scrivere dati. Questo modello di sviluppo presenta indubbi vantaggi quando si tratta di performance e usabilità ma ha come controindicazione un utilizzo massiccio di codice JavaScript sia dal punto di vista infrastrutturale sia dal punto di vista del codice di business.

Per ridurre al minimo il codice infrastrutturale necessario a creare una SPA, sono nati negli anni diversi framework e librerie, come Angular, ReactJS, Vue.JS e Aurelia. Poiché i primi due sono attualmente i più diffusi, Microsoft ha deciso di integrarli in Visual Studio, offrendo un template già pronto e l'integrazione con strumenti di sviluppo come, per esempio, NPM. Nel corso del capitolo parleremo sia di Angular sia di ReactJS, ma prima di iniziare ad affrontarli, approfondiamo il concetto di SPA.

Introduzione alle Single Page Application

Come abbiamo detto nell'introduzione del capitolo, una SPA è un'applicazione che gira interamente sul client e che sfrutta il server solo per recuperare i file JavaScript, CSS e HTML e per recuperare i dati. Questo significa che quando un utente digita l'url dell'applicazione, il server

risponde con tutti i file necessari affinché l'applicazione possa essere renderizzata sul browser e possa essere navigata senza ricorrere a ulteriori richieste al server, se non per recuperare e persistere i dati visualizzati nelle pagine effettuando chiamate AJAX a delle WebAPI.

Questo significa che se la nostra applicazione è composta da 20 pagine, alla prima richiesta il server invia una pagina HTML (detta master) contenente i riferimenti a tutti i file JavaScript e CSS necessari a renderizzare le 20 pagine. Il codice HTML delle 20 pagine è già incluso nei file JavaScript, velocizzando così il processo di renderizzazione.

Quando il browser ha finito di scaricare i file necessari, il codice JavaScript parte e identifica quale sia la home della nostra applicazione, ne prende il contenuto HTML e lo attacca al tag `body` della pagina master. Quando l'utente clicca su un url o compie un'altra azione che comporta la navigazione verso un'altra pagina, il codice JavaScript intercetta questa navigazione, identifica quale sia la pagina verso cui stiamo navigando, ne recupera il contenuto HTML, svuota il tag `body` della pagina master e lo riempie con il contenuto della nuova pagina. Questo processo di navigazione non comporta un post della pagina al server o una nuova richiesta al server, in quanto viene tutto completamente gestito lato client.

> *Da questa spiegazione emerge che dal server scarichiamo una sola pagina HTML. Questo è il motivo per cui questo genere di applicazioni viene chiamato Single Page Application.*

Ogni pagina che viene caricata nel master ha del codice JavaScript che la gestisce e con il quale può essere messa in binding. Questo codice è anche responsabile di invocare via AJAX il server qualora la pagina necessiti di visualizzare dati per essere renderizzata correttamente o necessiti di persistere dati. La Figura 21.1 mostra il funzionamento di una SPA.

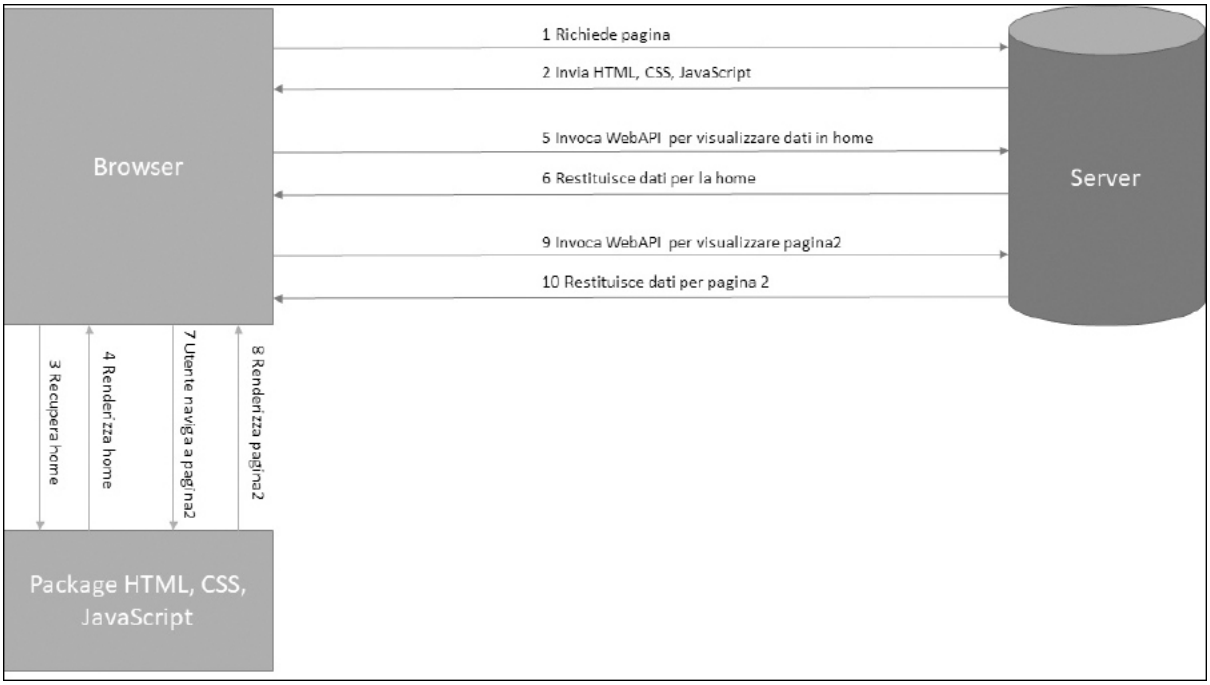


Figura 21.1 – L'interazione tra client e server avviene solo per la richiesta iniziale e per recuperare i dati per le pagine. Il resto avviene in locale.

Da questa breve introduzione emergono alcune importanti considerazioni. La prima è che le performance lato client delle SPA sono notevolmente superiori rispetto a quelle delle applicazioni ibride, poiché la navigazione avviene lato client senza alcun contatto col server e quindi senza alcuna latenza.

La seconda considerazione è che anche il server beneficia di migliori performance. Una volta inviati al client i file necessari, l'unica interazione che questo ha con il client è per lo scambio di dati via WebAPI. Questo significa che ci sono meno richieste al server e che queste richieste riguardano, nella maggior parte dei casi, solo lo scambio di dati. Rispetto alle applicazioni ibride che prevedono un continuo pattern di richiesta/risposta con il server con scambio di codice HTML, la quantità di dati che viaggia sulla rete è minore.

La terza considerazione è che anche l'usabilità delle SPA è notevolmente superiore a quella delle applicazioni ibride. Poiché tutto è gestito lato client, l'utente non ha l'effetto di ricaricamento della pagina a ogni richiesta e possiamo anche usare animazioni per passare da una pagina all'altra. A

tutti gli effetti, all'utente sembra di utilizzare un'applicazione nativa piuttosto che un'applicazione web.

Come sempre, ci sono pro e contro nello sviluppare una SPA rispetto a un'applicazione ibrida. Come si può immaginare, il codice JavaScript per gestire la parte infrastrutturale (intercettazione degli eventi di navigazione, rendering delle pagine nella master, caricamento dei file JavaScript su richiesta e non allo startup e altro ancora) è piuttosto corposo e complesso così come può esserlo anche quello di business. Nonostante i notevoli miglioramenti apportati al linguaggio negli ultimi anni, JavaScript non è ancora un linguaggio che si sposa molto bene in progetti di una certa entità e richiede molta disciplina agli sviluppatori. Gli strumenti di sviluppo sono notevolmente migliorati ma non siamo ancora ai livelli degli strumenti presenti per linguaggi come C#, e quindi, anche sotto questo punto di vista, ci troviamo svantaggiati nello sviluppare SPA.

Come sempre, la scelta tra un modello di sviluppo SPA e un modello ibrido dipende dalle conoscenze tecniche del team, dalle richieste del cliente e dai costi e dalle tempistiche. Se in un team ci sono molti sviluppatori JavaScript, probabilmente una SPA è la soluzione migliore, mentre se si hanno più sviluppatori C#, la scelta di creare un'applicazione ibrida è probabilmente la strada più sicura.

Se si decide di creare SPA, non si può prescindere dall'utilizzo di un framework che ne semplifichi lo sviluppo. Come abbiamo detto nell'introduzione, negli anni sono nati diversi framework che si occupano di tutti gli aspetti infrastrutturali, lasciando a noi il solo compito di configurare l'applicazione e scrivere il codice di business. Grazie a questi framework, il codice necessario a creare una SPA è notevolmente ridotto rispetto a quanto dovremmo scriverne se non li utilizzassimo. Il primo framework che prendiamo in considerazione è Angular.

Angular

Angular è il framework prodotto da Google per creare SPA. Angular è il successore di AngularJS, con cui però condivide parte della filosofia e poco altro. Angular è stato riscritto da zero utilizzando TypeScript come linguaggio di programmazione, con un'architettura modulare che permette di componentizzare la nostra applicazione in modo semplice.

Oltre al framework, Angular gode anche di un ecosistema che utilizza strumenti già consolidati nello sviluppo web (come NPM e WebPack) e strumenti ad-hoc come Angular-CLI. Grazie a questo ecosistema, lo sviluppo di una SPA basata su Angular è meno complesso rispetto a uno sviluppo effettuato con il suo predecessore.

Visual Studio offre un template che genera un'applicazione che sfrutta ASP.NET Core per le WebAPI e Angular per la UI attraverso il wizard di creazione di un'applicazione web, come è mostrato nella Figura 21.2. Al momento della stesura di questo libro, il template genera un'applicazione Angular 5 ma in un aggiornamento futuro di Visual Studio il template verrà aggiornato per generare un'applicazione sfruttando Angular 6.

Il progetto creato da Visual Studio genera lo scheletro di un'applicazione ASP.NET Core con un controller per le WebAPI che fornisce dati preimpostati e con la cartella ClientApp che contiene l'applicazione Angular, che è quella che ci interessa.

Per creare lo scheletro di quest'ultima, Visual Studio sfrutta Angular-CLI. Questo strumento, che fa parte del SDK di Angular ed è scaricato da NPM, permette di creare un'applicazione da zero e di aggiungervi componenti man mano che si sviluppa l'applicazione. Permette inoltre di compilare la nostra applicazione sia in debug sia in produzione, avviare un web server per il debug e altro ancora.

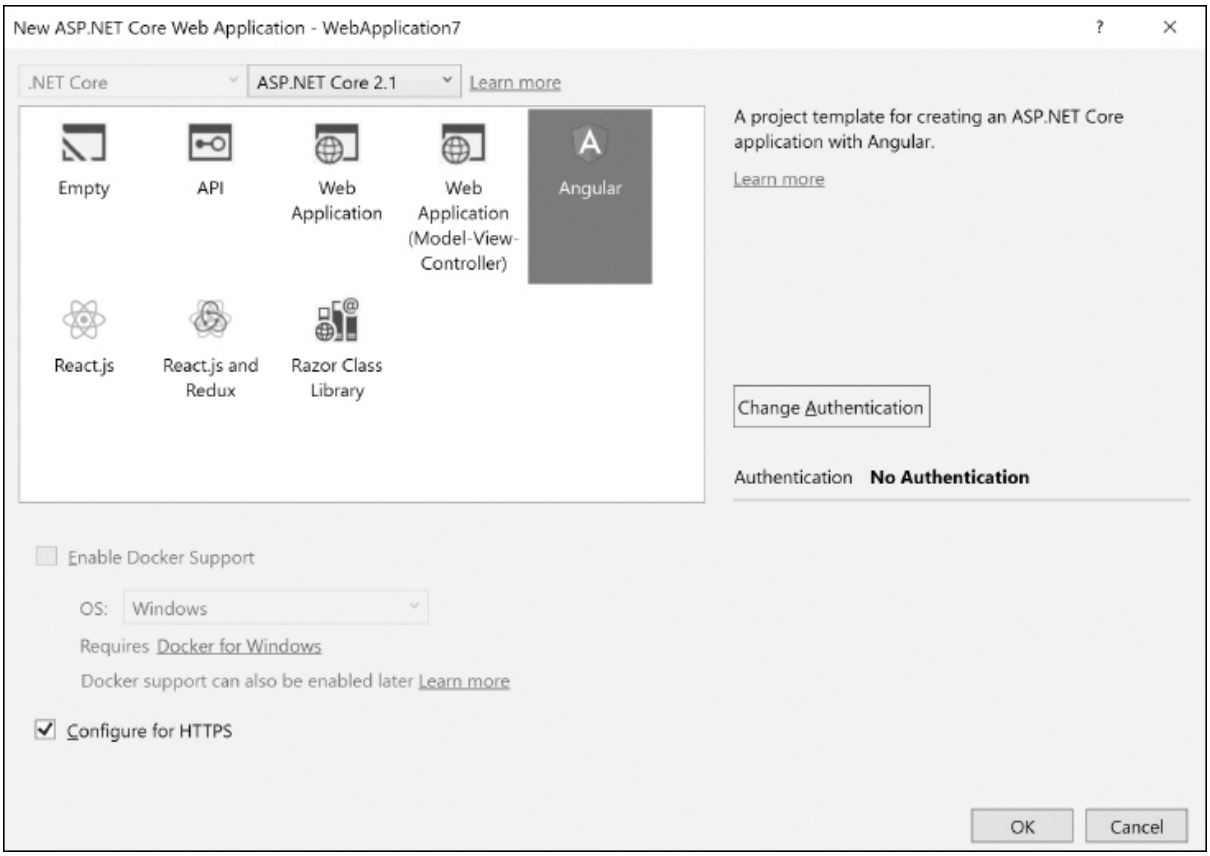


Figura 21.2 – Il wizard di creazione di un'applicazione web include l'opzione per Angular.

> *Accenneremo ad alcuni comandi della CLI nel corso del capitolo ma, per maggiori informazioni, si può consultare il sito di Angular-CLI, all'indirizzo: http://aspit.co/bpn.*

Una volta terminata la creazione del progetto, la finestra Solution Explorer appare come nella Figura 21.3.

Figura 21.3 – La finestra Solution Explorer mostra il codice di un'applicazione Angular appena creata tramite Visual Studio.

Nella root della cartella ClientApp troviamo i file necessari a configurare l'ambiente come il file `tsconfig.json`, che stabilisce le regole di compilazione di TypeScript, il file `tslint.json`, che specifica le regole di formattazione del codice, il file `package.json`, per stabilire strumenti e librerie da scaricare da NPM (tra cui quelle di Angular) e il file `.angular-cli.json`, per configurare Angular-CLI.

Il codice sorgente si trova nella cartella src, all'interno della quale troviamo il file `index.html`, che sarà la pagina HTML master, il file `main.ts`,

che è responsabile dell'inizializzazione dell'applicazione, il file `styles.css`, che contiene gli stili CSS e il file `polyfill.ts`, che contiene i polyfill necessari ad Angular a seconda del browser e della versione.

La cartella environments contiene i file `environment.ts` e `environment.prod.ts`, che specificano un oggetto JSON con i parametri dell'applicazione. Quando compiliamo in debug, viene usato il primo file mentre, se compiliamo per la produzione, viene usato il secondo file grazie a un parametro nel file `.angular-cli.json`.

La cartella assets contiene tutti i file necessari alla nostra applicazione per funzionare ed essere visualizzata correttamente come file JavaScript, CSS, immagini, font e così via. I file in questa cartella sono automaticamente inclusi nel pacchetto generato dalla compilazione grazie a un parametro impostato nel file `.angular-cli.json`.

Un'applicazione Angular è composta dai componenti elencati nella Tabella 21.1.

Tabella 21.1 – Componenti di un'applicazione Angular.

Componente	Descrizione
Component	Frammento di HTML e classe TypeScript che interagiscono per gestire una pagina dell'applicazione o una sua porzione
Service	Contiene il codice di business dell'applicazione. Tipicamente usata come interfaccia tra i component e le WebAPI
Pipe	Classe che permette di formattare l'output di una proprietà in binding (utile per formattare date, numeri e così via)
Directive	Classe che permette di aggiungere comportamenti a un component o un tag HTML
Module	Si tratta di un contenitore degli oggetti appena menzionati

Nella cartella app troviamo i vari componenti della nostra applicazione. La prima cosa che troviamo è il component `app.component` suddiviso in tre file con estensione .css, .html e .ts. Il file CSS specifica un file CSS che è valido solo all'interno del component (opzionale), il file HTML specifica il frammento di HTML che il component gestisce, mentre il file TS è la classe TypeScript con cui il frammento HTML comunica in binding.

`app.component` è il component lanciato dalla nostra applicazione in fase di startup e corrisponde quindi alla home. Angular prende il codice HTML del component e lo aggancia non al tag `body` della pagina index.html bensì all'interno del tag `<app-root>` visibile nel prossimo esempio.

Esempio 21.1

```
<body>
  <app-root>Loading...</app-root>
</body>
```

Ora che il frammento di HTML è stato attaccato alla pagina, possiamo metterlo in binding con il suo component. Nell'Esempio 21.2 vediamo sia il file HTML sia il file TS e notiamo come questi comunicano e quali funzionalità espongono.

Esempio 21.2 - TypeScript

```
import { Component } from ´@angular/core´;
@Component({
  selector: ´app-root´,
  templateUrl: ´./app.component.html´,
  styleUrls: [´./app.component.css´]
})
export class AppComponent {
  title = ´app´;
}
```

Esempio 21.2 - HTML

```
<div class=´container-fluid´>
  <div class=´row´>
    <div class=´col-sm-3´>
      <app-nav-menu></app-nav-menu>
    </div>
    <div class=´col-sm-9 body-content´>
      <router-outlet></router-outlet>
    </div>
  </div>
</div>
```

Questo esempio contiene molte funzionalità di Angular. Ma andiamo per passi, cominciando dal file TypeScript. Questo file contiene una semplice

classe che ha una proprietà di nome app e che viene agganciata ad Angular grazie al decorator @Component che fa parte del framework e che viene importato nel file grazie all'utilizzo di import.

Il decorator espone le proprietà selector, che specifica il tag HTML associato al component, templateUrl, che specifica il nome del file HTML legato alla classe (template), e styleUrls, che specifica eventuali file CSS da associare al template. È grazie alla proprietà selector che, in fase di startup, Angular è in grado di collegare il tag <app-root> nel file index.html al component e utilizzarlo come home dell'applicazione.

Il template contiene esclusivamente codice HTML oltre a due tag sconosciuti: <app-nav-menu> e <router-outlet>. Il primo tag renderizza al suo interno un altro component che ha specificato il valore app-nav-menu nella proprietà selector del suo decorator. Questo component si trova nella cartella nav-menu e contiene il menu di navigazione. Il secondo è un tag Angular che specifica l'area in cui inserire il codice HTML dei component che carichiamo quando navighiamo nell'applicazione. Da questo deriva che app.component contiene il layout generale della nostra applicazione, in cui è compreso anche il menu.

Visto che app.component non è una vera e propria pagina dell'applicazione ma un contenitore, dobbiamo avere a disposizione un altro component verso cui eseguire la navigazione quando si carica l'app. Questo mapping lo specifichiamo nel modulo app.module, all'interno del quale specifichiamo i component della nostra applicazione e il routing, come viene mostrato nel seguente codice.

Esempio 21.3

```
@NgModule({
  declarations: [
    AppComponent,
    NavMenuComponent,
    HomeComponent,
    CounterComponent,
    FetchDataComponent
  ],
  imports: [
    BrowserModule.withServerTransition({ appId: ´ng-cli-universal´ }),
    HttpClientModule,
    FormsModule,
    RouterModule.forRoot([
      { path: ´´, component: HomeComponent, pathMatch: ´full´ },
      { path: ´counter´, component: CounterComponent },
      { path: ´fetch-data´, component: FetchDataComponent },
```

```
    ])
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Un module è una classe che viene collegata ad Angular grazie al decorator `@NgModule`. Questo decorator espone la proprietà `declarations`, tramite la quale specifichiamo i component del module, la proprietà `imports`, tramite la quale specifichiamo quali altri moduli importiamo nel nostro per poterne utilizzare le classi, `providers` che contiene i service e `bootstrap` che specifica il component di startup (da utilizzare solo nel modulo principale).

Poiché la nostra applicazione naviga tra pagine, importiamo il module `RouterModule` e, tramite il suo metodo `forRoot`, creiamo un mapping tra url e component da caricare. Nel nostro caso, quando chiediamo la root del sito, viene caricato `app-component` e poi si naviga verso il component `HomeComponent`. Quando l'utente naviga verso l'url `counter`, viene caricato `CounterComponent` e, se si naviga verso l'url `fetch-data`, viene caricato `FetchDataComponent`.

Quest'ultimo component è interessante, in quanto mostra perfettamente come mettere in binding il template con i dati nel component presi da una WebAPI. Cominciamo dal codice del component.

Esempio 21.4

```
@Component({
  selector: ´app-fetch-data´,
  templateUrl: ´./fetch-data.component.html´
})
export class FetchDataComponent {
  public forecasts: WeatherForecast[];

  constructor(http: HttpClient, @Inject(´BASE_URL´) baseUrl: string) {
    http.get<WeatherForecast[]>(baseUrl + ´api/SampleData/WeatherForecasts´)
      .subscribe(result => this.forecasts = result);
  }
}
interface WeatherForecast {
  dateFormatted: string;
  temperatureC: number;
  temperatureF: number;
  summary: string;
}
```

l

l component ha una proprietà `forecasts` che è un array di oggetti di tipi di tipo `WeatherForecast`. Nel costruttore della classe, accettiamo in input due parametri: uno di tipo `HttpClient` che rappresenta l'oggetto da usare per effettuare chiamate AJAX verso le WebAPI e l'altro di tipo string, che rappresenta l'url di base delle WebAPI. Questi parametri vengono iniettati nel costruttore del component dal motore di dependency injection di Angular.

Per recuperare i dati, usiamo il metodo `get` specificando tramite parametro generico il tipo di oggetto restituito dal server e passando in input l'url del servizio. Questo metodo restituisce un oggetto di tipo `Observable` (classe facente parte della libreria RxJS da cui Angular dipende) e noi usiamo il metodo `subscribe` per ricevere le notifiche della risposta dal server. Quando il risultato arriva, viene invocato il callback passato in input al metodo `subscribe`, all'interno del quale valorizziamo la proprietà `forecasts` con il risultato della chiamata.

> *La classe HttpClient espone anche i metodi post, put, delete e altri ancora per effettuare ogni tipo di chiamata HTTP.*

Una volta recuperati i dati dal servizio, dobbiamo visualizzarli sulla UI e questo è compito del template tramite il codice dell'Esempio 21.5.

Esempio 21.5

```
<p *ngIf="!forecasts"><em>Loading...</em></p>

<table class='table' *ngIf="forecasts">
  <thead>
    <tr>
      <th>Date</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let forecast of forecasts">
      <td>{{ forecast.dateFormatted }}</td>
    </tr>
  </tbody>
</table>
```

La prima caratteristica di questo esempio sono le direttive `ngIf` e `ngFor` di Angular. La prima renderizza o no un tag se la condizione della direttiva è

vera o falsa, mentre la seconda ripete il tag tante volte quando gli elementi contenuti nell'array che si sta ciclando nell'espressione passata in input alla direttiva. In questo esempio, la proprietà `forecasts` è `undefined` finché il servizio non risponde, quindi viene mostrato il messaggio di caricamento finché il servizio non risponde e poi, una volta che la proprietà `forecasts` è valorizzata con la risposta del servizio, il messaggio viene nascosto e viene mostrata la tabella. Tramite `ngFor` per ogni oggetto dell'array replichiamo il template, sfruttando la sintassi che prevede le parentesi graffe doppie per specificare il binding, come abbiamo già visto nel Capitolo 19 con Vue.js. Il risultato è visibile nella Figura 21.4.

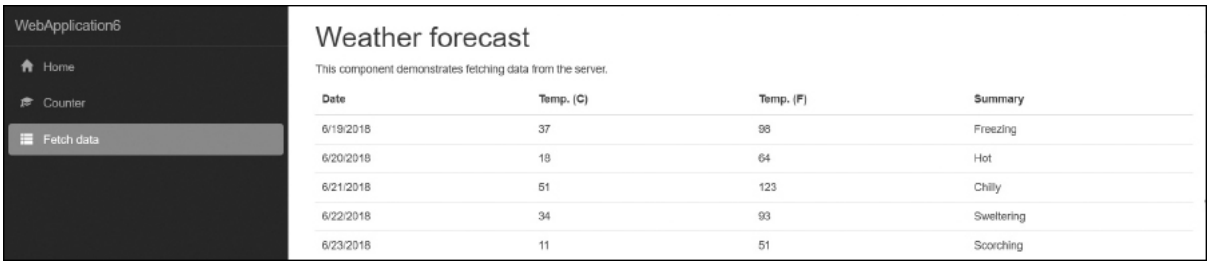


Figura 21.4 – L'applicazione mostra i dati ricevuti dal servizio.

Questo esempio mostra chiaramente come la separazione dei compiti tra component e template sia netta e come questi si parlino via binding. Tuttavia, abbiamo visto solo un rovescio della medaglia cioè il binding dal component verso il template, ma possiamo fare anche l'inverso, ovvero far reagire il component a un evento sulla UI come il click di un pulsante. Questo esempio è mostrato nel `CounterComponent` e ne mostriamo un estratto nel prossimo codice.

Esempio 21.6 - TypeScript

```
export class CounterComponent {
  public currentCount = 0;

  public incrementCounter() {
    this.currentCount++;
  }
}
```

Esempio 21.6 - HTML

```
<p>Current count: <strong>{{ currentCount }}</strong></p>
<button (click)="incrementCounter()">Increment</button>
```

Grazie all'utilizzo delle parentesi tonde, intorno alla direttiva `click` specifichiamo ad Angular che stiamo mettendo in binding un evento della UI con un metodo sul component. Successivamente, il metodo del component aggiorna il valore della proprietà `currentCount`, che essendo in binding viene aggiornata sulla UI.

Se vogliamo vedere in azione l'applicazione, tutto quello che dobbiamo fare è lanciarla con o senza debug e navigare. Quando lanciamo l'applicazione, il compilatore di Angular lancia prima il compilatore TypeScript per convertire il codice in JavaScript, poi compila il codice HTML e CSS per includerlo nei file JavaScript e, infine, copia questi file, la cartella assets e il file `index.html` in una cartella dist a cui il web server punta. Se vogliamo generare una build da mettere in produzione, ci basta entrare nella riga di comando e digitare il comando `ng build --prod`, che esegue gli stessi passi con la sola differenza di ottimizzare al massimo il codice per renderlo il più piccolo possibile.

In questa sezione abbiamo visto come creare un progetto con Angular e ASP.NET Core, partendo da zero, abbiamo visto come il progetto è strutturato e come funzionano binding e navigazione in Angular. Ovviamente in Angular c'è molto di più, come le form, la creazione di controlli custom e librerie, la creazione Progressive Web Application e altro ancora. Per ovvi motivi di spazio non ci è possibile approfondire Angular, per cui rimandiamo alla documentazione, che è disponibile all'indirizzo: http://aspit.co/bpo.

ReactJS

ReactJS è la libreria targata Facebook per creare SPA. Così come Angular, questa ibreria è largamente usata dagli sviluppatori web e per questo motivo Microsoft ha deciso di includerla all'interno di Visual Studio, sempre utilizzando il wizard di creazione di un progetto web e selezionando nella finestra della Figura 21.2 l'opzione React.js al posto di Angular. Il template dietro questa opzione genera un'applicazione con le stesse funzionalità di quella generata per Angular, quindi il risultato alla fine del processo di

creazione è un progetto che ha lo stesso controller per le WebAPI e la cartella ClientApp che è come quella visibile nella Figura 21.5.

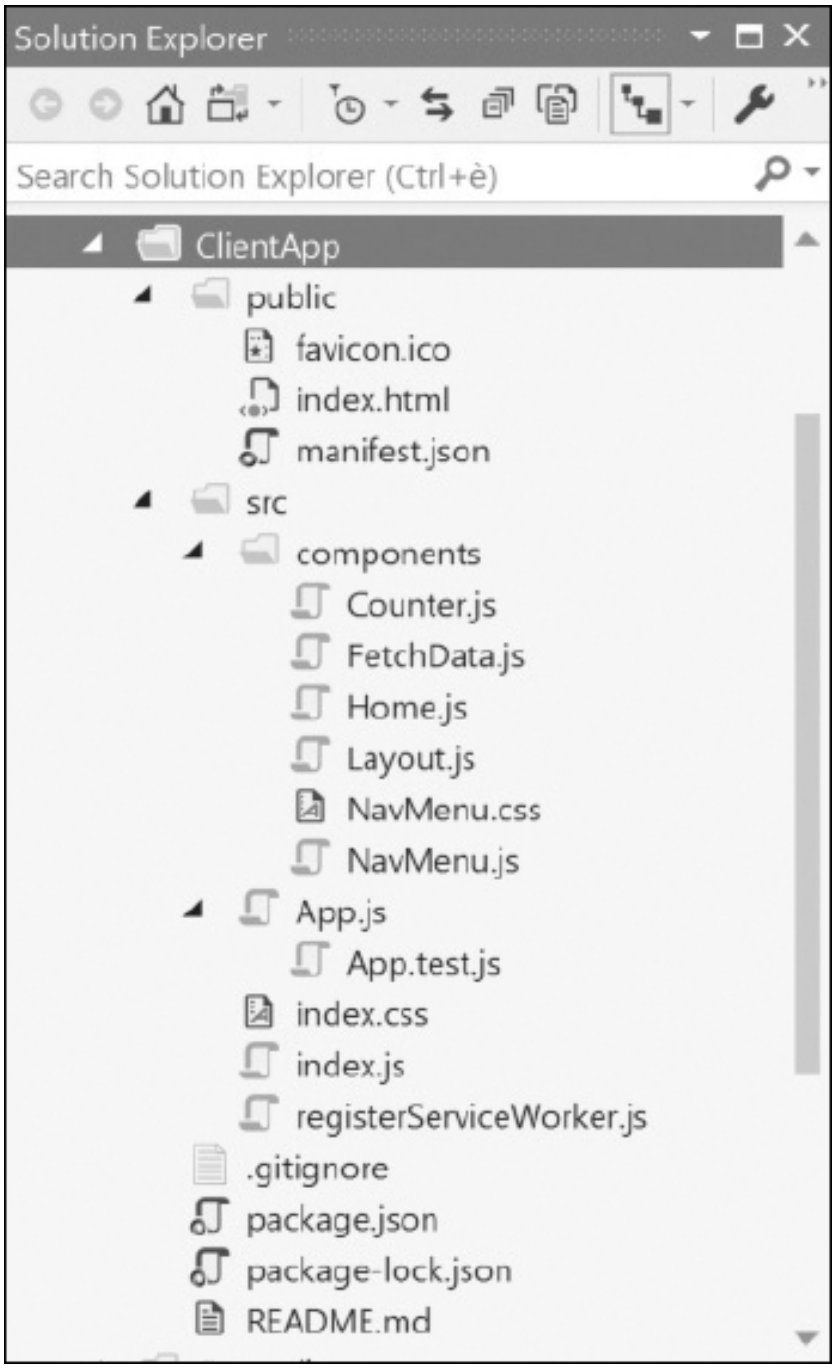


Figura 21.5 – La finestra Solution Explorer mostra l'applicazione ReactJS appena creata.

Nella root di ClientApp troviamo il file `package.json` che contiene i riferimenti agli strumenti e alle librerie da scaricare da NPM mentre nella cartella public troviamo alcuni file tra cui `index.html`, che rappresenta il master e che contiene nel tag `<body>` solo un tag `<div>` con id `root`, all'interno del quale girerà l'applicazione.

La cartella src contiene il vero codice dell'applicazione, a partire dal file `index.js`, che contiene il codice di startup mostrato nell'Esempio 21.7.

Esempio 21.7

```
const baseUrl =
  document.getElementsByTagName(´base´)[0].getAttribute(´href´);
const rootElement = document.getElementById(´root´);

ReactDOM.render(
<BrowserRouter basename={baseUrl}>
  <App />
</BrowserRouter>,
rootElement);
```

Il metodo `render` della classe `ReactDOM` renderizza il codice HTML al suo interno, sfruttando la sintassi JSX. Questa è una grossa differenza rispetto ad Angular, dove il codice HTML è in un template a parte. `<BrowserRouter>` è un tag che rappresenta un componente di ReactJS, all'interno del quale renderizzare le pagine dell'applicazione durante la navigazione, mentre `<app>` è un tag che rappresenta un componente custom dell'applicazione contenuto nel file `app.js`. Il secondo parametro è l'oggetto HTML all'interno del quale renderizzare il codice HTML passato come primo parametro.

Il file `app.js` rappresenta un componente in ReactJS e si tratta di una classe che estende la classe base `Component` di cui esegue l'override del metodo `render`, che restituisce il codice HTML del componente. Il metodo `render` del componente `App` è quello dell'Esempio 21.8.

Esempio 21.8

```
export default class App extends Component {
  displayName = App.name

  render() {
    return (
      <Layout>
        <Route exact path=´/´ component={Home} />
```

```
          <Route path=´/counter´ component={Counter} />
          <Route path=´/fetchdata´ component={FetchData} />
        </Layout>
      );
    }
  }
```

Il tag `<Layout>` rappresenta un altro componente dell'applicazione che al suo interno prende la lista delle rotte dell'applicazione e il componente relativo a ogni rotta. `Layout` comprende il layout HTML dell'applicazione come l'intestazione, il menu e il punto in cui la pagina specificata dal routing deve essere renderizzata.

Anche in questo caso, le pagine sono rappresentate da due componenti `FetchData` e `Counter`. Il componente `FetchData` esegue la chiamata al server e mostra i dati a video. Quindi, è quello che più ci interessa per quanto riguarda l'interazione con ASP.NET Core. Vediamo il suo codice nell'Esempio 21.9.

Esempio 21.9

```
export class FetchData extends Component {
  constructor(props) {
    super(props);
    this.state = { forecasts: [], loading: true };

    fetch(´api/SampleData/WeatherForecasts´)
      .then(response => response.json())
      .then(data => {
        this.setState({ forecasts: data, loading: false });
      });
  }

  static renderForecastsTable(forecasts) {
    return (
      <table className=´table´>
        <thead>
          <tr>
            <th>Date</th>
          </tr>
        </thead>
      <tbody>
        {forecasts.map(forecast =>
          <tr key={forecast.dateFormatted}>
            <td>{forecast.dateFormatted}</td>
          </tr>
        )}
      </tbody>
    </table>
   );
  }

  render() {
    let contents = this.state.loading
      ? <p><em>Loading...</em></p>
```

```
        : FetchData.renderForecastsTable(this.state.forecasts);
      return (
        <div>
          <h1>Weather forecast</h1>
          <p>This component demonstrates fetching data from the server.</p>
          {contents}
        </div>
      );
    }
  }
```

Ogni componente in ReactJS ha uno stato accessibile tramite la proprietà `state`, manipolabile tramite il metodo `setState` e messo in binding con il codice HTML. In questo caso, nel costruttore impostiamo lo stato con un oggetto che ha la proprietà `loading`, che specifica se i dati sono stati caricati o no, e la proprietà `forecasts`, che contiene i dati recuperati dal server. Successivamente, invochiamo il metodo `fetch` passando in input l'url della WebAPI. Quando il server risponde, impostiamo la proprietà `forecasts` con i dati restituiti dal server e la proprietà `loading` a `true` per indicare che il caricamento è terminato.

Il metodo `render` renderizza un messaggio di attesa finché la proprietà `loading` è `false` e la tabella con i dati, una volta caricati. Il codice HTML è interpolato con codice JavaScript attraverso l'utilizzo delle parentesi graffe singole. Questo significa che, a differenza di Angular, non abbiamo direttive per simulare istruzioni `if`, `for` o altro ancora, ma utilizziamo direttamente il codice JavaScript. Per alcuni sviluppatori la sintassi JSX per descrivere il codice HTML è più chiara, mentre per altri lo è meno. Questo pesa molto sulla scelta tra un framework e l'altro.

Anche in questo caso, abbiamo visto solamente il binding dal componente verso la UI, ma ReactJS permette ovviamente il contrario cioè di invocare un metodo sul componente a seguito di un evento sulla UI. Anche in questo caso, ci viene in aiuto la sintassi che usa le parentesi graffe singole.

Esempio 21.10

```
export class Counter extends Component {
  constructor(props) {
    super(props);
    this.state = { currentCount: 0 };
    this.incrementCounter = this.incrementCounter.bind(this);
  }
```

```
    incrementCounter() {
      this.setState({
        currentCount: this.state.currentCount + 1
      });
    }

    render() {
      return (
        <div>
          <p>Current count: <strong>{this.state.currentCount}</strong></p>
          <button onClick={this.incrementCounter}>Increment</button>
        </div>
      );
    }
  }
```

Anche in questo caso, la trattazione di ReactJS è limitata allo startup, al disegno della UI, al binding e all'interfacciamento con il server per lavorare con i dati. Ovviamente c'è un universo di funzionalità che per ovvi motivi non possiamo coprire ma che può essere approfondito direttamente sul sito di ReactJS, disponibile all'url: `http://aspit.co/bpq`.

Conclusioni

Le SPA sono un paradigma di sviluppo che sta prendendo sempre più piede nel mondo web. Ormai, quando si inizia un nuovo progetto, questo modello è una scelta da tenere sempre in considerazione in virtù del fatto che la sua maturazione è ormai a un livello tale da renderlo equiparabile in molti casi al modello ibrido.

La quantità e la qualità dei framework a disposizione è tale da non far rimpiangere la mancanza di un linguaggio fortemente tipizzato come C#. Inoltre, le specifiche di HTML, CSS e JavaScript sono in continua evoluzione il che continua ad aprire nuovi scenari in cui le SPA possono essere un player insostituibile, come nel caso delle Progressive Web Application.

Ovviamente non è tutto oro quel che luccica, quindi la scelta tra il modello SPA e il modello ibrido deve essere sempre attentamente valutata.

Con questo argomento abbiamo terminato la nostra trattazione relativa ad argomenti di sviluppo con ASP.NET Core, ma non è finita qui. Le nostre applicazioni, infatti, devono anche essere messe in produzione. Nel prossimo capitolo ci occuperemo proprio di questi argomenti.

22

Configurare e pubblicare un'applicazione ASP.NET Core

Nel corso del libro abbiamo affrontato tutti i temi principali relativi allo sviluppo di un'applicazione web, partendo dai concetti di MVC, arrivando alla definizione delle view, fino a parlare dei dati e di come localizzarli, esporli, proteggerli e consumarli. Tuttavia, nonostante tutti questi concetti siano risultati utili alla costruzione di una vera e propria applicazione, non possono essere messi in pratica e testati effettivamente fino a quando non si arriva alla fase del rilascio, in modo che chiunque possa vedere il prodotto realizzato.

Tipicamente, in passato, distribuire un'applicazione ASP.NET significava preparare un'ambiente con macchine Windows Server e Internet Information Server (IIS). Inoltre, era quasi dato per scontato che la versione del .NET Framework utilizzata dal team di sviluppo fosse la stessa disponibile nell'ambiente di produzione, soprattutto considerati i lunghi periodi tra il rilascio di una versione e la successiva. A oggi, però, sono cambiati tanti aspetti:

- Tempistiche di rilascio: i tempi che intercorrono tra la fase di sviluppo e il rilascio in produzione sono notevolmente ridotti per via di pratiche come continuous integration e continuous deployment.

- Ambienti multipli: è fondamentale rilasciare un prodotto che sia testato – possibilmente da più team – e funzionante (sviluppo, QA di diversi livelli) ma, soprattutto per via dei tempi ristretti, è bene

testare le applicazioni in ambienti che non siano produzione (per evitare potenziali problematiche) ma che siano piuttosto simili per effettuare simulazioni di upgrade e downgrade.

- L'hardware e il software: .NET Core, al contrario di .NET Framework, è in grado di eseguire le applicazioni anche in ambienti Linux e macOS, quindi non è più possibile dare per scontata la presenza di Windows Server e di IIS; inoltre, il framework potrebbe non essere installato fisicamente sulla macchina ma distribuito con l'applicazione stessa.

- Tipologia di lavoro: il team di lavoro moderno e ideale si sta sempre di più spostando verso il mondo agile e DevOps, quindi diventa fondamentale avere conoscenze sia di sviluppo software sia di gestione e manutenzione (operations) dell'infrastruttura sulla quale viene eseguito il software, per questo sono emersi temi fondamentali come i container e il cloud.

All'interno di questo capitolo affronteremo proprio il concetto relativo al deployment e al delivery delle applicazioni e parleremo nel dettaglio di come le novità introdotte da .NET Core (e ASP.NET Core) aiutino a risolvere le problematiche sollevate dagli aspetti evidenziati. Prima di procedere al rilascio dell'applicazione, però, è lecito domandarsi in quale ambiente l'applicazione stessa verrà distribuita: di fatto, è possibile prevedere un cambio del comportamento sulla specifica dell'ambiente.

Modificare il comportamento secondo l'ambiente di destinazione

Come abbiamo già descritto nei primi capitoli di questo libro, ci sono casi in cui il comportamento dell'applicazione può essere diverso a seconda dell'ambiente in cui viene eseguito: l'ambiente di sviluppo (o `Development`) potrebbe prevedere un logging molto avanzato e continuo su ogni singola chiamata, mentre l'ambiente di produzione potrebbe avere solo un logging sommario per non rallentare l'esecuzione, così come nell'ambiente di controllo qualità si potrebbe aver bisogno di un

logging di tipo misto, in cui ha più senso verificare il flusso delle operazioni effettuate dagli utenti rispetto al sapere il dettaglio di risposta di ogni singola riga di codice. In produzione si vorrà avere la minification o, ancora, ci saranno endpoint differenti in base all'ambiente per richiamare una WebAPI oppure un background service. ASP.NET Core mette a disposizione gli environment, che permettono proprio di distinguere, a livello di codice, l'ambiente di esecuzione a seconda di quanto viene definito dagli sviluppatori oppure da operations direttamente sulla macchina fisica.

Come abbiamo già avuto modo di vedere in precedenza, gli environment di default sono `Development`, `Staging` e `Production` che vengono esposti direttamente dal framework tramite l'interfaccia `IHostingEnvironment` e, in particolare, tramite la proprietà `EnvironmentName`, ma, poiché ci sono scenari che richiedono configurazioni e ambienti molto più complessi, è anche possibile andare a creare ambienti personalizzati.

Esempio 22.1

```
public class MyClass
{
  public MyClass(IHostingEnvironment en)
  {
    // controlliamo l´ambiente
    if (en.IsEnvironment(˝MyCustomEnvironment˝)
    {
      // l´ambiente è identificato, eseguiamo operazioni specifiche...
    }
  }
}
```

Nell'Esempio 22.1 si può notare come sia possibile identificare un ambiente personalizzato per poter eseguire le operazioni specifiche relative a quell'environment. Non tutta la configurazione, però, è strettamente legata ai servizi che devono essere erogati (come, per esempio, il cambio di URL tra ambiente di sviluppo e di produzione), ma può anche essere relativa all'infrastruttura su cui viene eseguita, come per esempio l'uso di Kestrel piuttosto che IIS per il web server, l'entrypoint dell'applicazione stessa o, ancora, l'uso di variabili

d'ambiente personalizzate. Tutte queste opzioni possono essere specificate all'interno del file `launchSettings.json` nella cartella `Properties` del progetto, come mostrato nell'Esempio 22.2.

Esempio 22.2

```
{
  "iisSettings": {
    "windowsAuthentication": false,
      "anonymousAuthentication": true,
      "iisExpress": {
        "applicationUrl": "http://localhost:43421",
        "sslPort": 44300
      }
    },
    "profiles": {
      "IIS Express": {
        "commandName": "IISExpress",
        "launchBrowser": true,
        "environmentVariables": {
          "ASPNETCORE_ENVIRONMENT": "Development"
        }
      },
      "Capitolo22": {
        "commandName": "Project",
        "launchBrowser": true,
        "applicationUrl": "https://localhost:5001;http://localhost:5000",
        "environmentVariables": {
          "ASPNETCORE_ENVIRONMENT": "Development",
          "ASPNETCORE_CUSTOM_VALUE": "Custom",
      }
    }
  }
}
```

Tra le variabili d'ambiente esposte dall'Esempio 22.2 si può notare la sola `ASPNETCORE_ENVIRONMENT` che rappresenta proprio l'`EnvironmentName` di `IHostingEnvironment` per il profilo "IIS Express", mentre per il profilo "Capitolo22" è stata aggiunta anche una variabile custom. Tra le altre proprietà esposte da questo file di configurazione è bene evidenziare `commandName` poiché, durante l'esecuzione del comando `dotnet run`, verrà fatto il discovery di tutti i profili e quindi verrà scelto il primo con il valore impostato a `Project`. I profili elencati all'interno di questo file sono anche utilizzati da Visual Studio per far partire l'applicazione web nel modo corretto, come viene illustrato nella Figura 22.1.

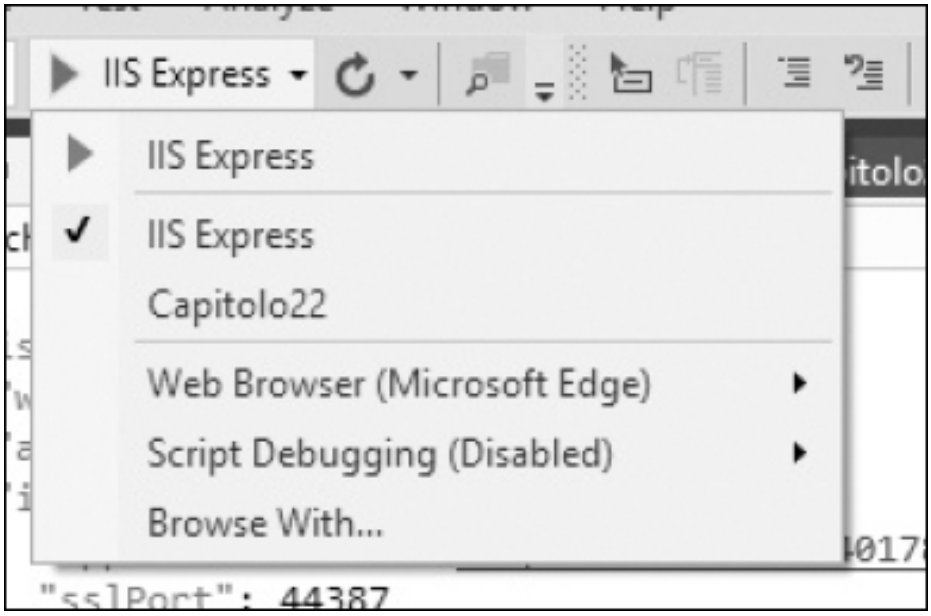


Figura 22.1 – I profili creati all'interno del file `launchSettings.json` vengono proposti da Visual Studio per l'esecuzione dell'applicazione web.

Il comando `dotnet run`, nello specifico, esegue questa serie di operazioni in sequenza, che vengono mostrate nella Figura 22.2: recupera il file `launchSettings.json` per capire quale profilo avviare, quindi legge i valori delle variabili d'ambiente e li sostituisce a quelli eventualmente definiti tramite variabili d'ambiente a livello di sistema, poi recupera il valore dell'environment, esegue il `Main` della classe `Program` che avvia il web server e apre le porte necessarie e, infine, carica la configurazione opportuna.

```
Windows PowerShell
PS C:\book\Capitolo22> dotnet run
Uso delle impostazioni di avvio di C:\book\Capitolo22\Properties\launchSettings.json...
info: Microsoft.AspNetCore.DataProtection.KeyManagement.XmlKeyManager[0]
      User profile is available. Using 'C:\Users\Matteo\AppData\Local\ASP.NET\DataProtection-Keys'
as key repository and Windows DPAPI to encrypt keys at rest.
Hosting environment: Development
Content root path: C:\book\Capitolo22
Now listening on: https://localhost:5001
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
```

Figura 22.2 – L'esecuzione del comando `dotnet run` forza la lettura delle impostazioni di avvio.

La variabile d'ambiente `ASPNETCORE_ENVIRONMENT` può essere impostata a diversi livelli, in modo dipendente dal sistema operativo e dalla durata prevista. Per avere un'impostazione temporanea a livello di finestra/esecuzione corrente, su Windows, per esempio, può essere impostata tramite i comandi `setx` e `$Env` rispettivamente per Command Line e PowerShell, mentre per macOS e distribuzioni Linux può essere impostata tramite il comando `export`. Purtroppo, però, l'esecuzione di questi comandi è limitata, quindi in ambienti di produzione o in scenari in cui l'applicazione può andare in crash non è l'ideale, dato che riavviandosi perderebbe l'ambiente precedentemente impostato. Per questo è necessario impostare la variabile direttamente a livello di sistema operativo, tramite il `bash_profile` di Linux e macOS, oppure tramite le variabili d'ambiente di Windows, come illustrato nella Figura 22.3.

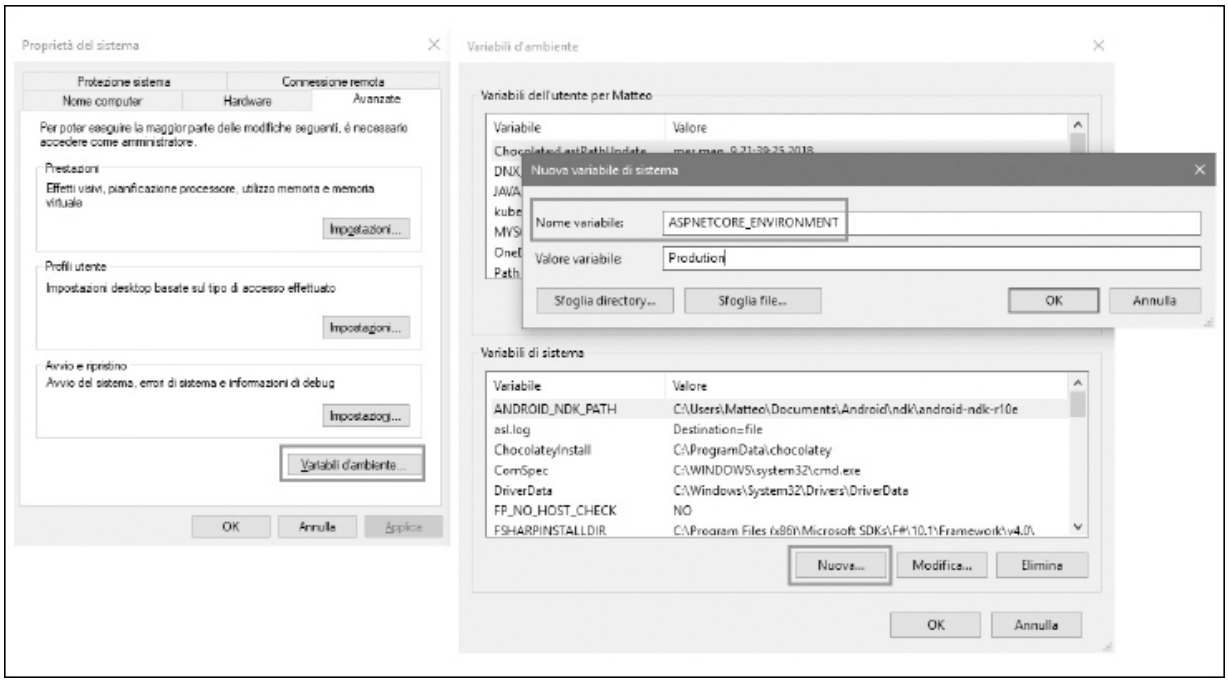


Figura 22.3 – L'impostazione di una variabile di sistema su sistema operativo Windows avviene tramite il menù Proprietà di sistema ->

Variabili d'ambiente -> Nuova variabile di sistema.

Qualora l'applicazione venga eseguita in ambiente Windows con IIS, allora è anche possibile specificare l'environment e le altre variabili d'ambiente tramite il `web.config`, come dimostra l'Esempio 22.3.

Esempio 22.3

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <handlers>
      <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModule"
resourceType="Unspecified" />
      </handlers>
      <aspNetCore processPath="dotnet"
                  arguments=".\Capitolo22.dll"
                  stdoutLogEnabled="false"
                  stdoutLogFile=".\logs\stdout">
      <environmentVariables>
        <environmentVariable name="ASPNETCORE_ENVIRONMENT" value="Development" />
        <environmentVariable name="ASPNETCORE_CUSTOM_VALUE" value="Custom" />
      </environmentVariables>
    </aspNetCore>
  </system.webServer>
</configuration>
```

All'interno di un progetto vuoto ASP.NET Core MVC, la prima volta in cui si incontra l'uso della variabile `ASPNETCORE_ENVIRONMENT` è nel metodo `Configure` della classe `Startup`, in cui l'uso della pagina di errore dettagliato e di HSTS vengono aggiunti solamente per gli ambienti di sviluppo e produzione rispettivamente, come è illustrato nell'Esempio 22.4.

Esempio 22.4

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
  if (env.IsDevelopment())
  {
    app.UseDeveloperExceptionPage();
  }
  else
  {
    app.UseExceptionHandler("/Error");
```

```
      app.UseHsts();
    }
    app.UseMvc();
  }
```

Se però l'applicazione cresce e diventa via via più complessa, cresce il numero di environment oppure il numero di servizi e l'esecuzione degli stessi dipende dagli ambienti, può diventare molto complicato riuscire a gestire tutto il flusso di inizializzazione previsto dalla classe `Startup` solo con una serie di statement condizionali. Proprio per questo motivo, i metodi `Configure` e `ConfigureServices` in realtà sono solo un'astrazione dei veri nomi, rappresentati da `Configure{environment-name}` e `Configure{environment-name}Services`.

Esempio 22.5

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddMvc();
}
public void ConfigureDevelopmentServices(IServiceCollection services)
{
  services.AddAuthentication().AddMvc();
}
```

Come si può notare nell'Esempio 22.5, infatti, anziché usare un `if` per controllare se l'ambiente di esecuzione è `Development`, per aggiungere l'autenticazione si è preferito utilizzare la separazione tra i due metodi. Poiché la stessa cosa può avvenire per entrambi i metodi `Configure` e `ConfigureServices` per ogni environment configurato, si corre il rischio di aggiungere troppa complessità e illeggibilità alla classe di `Startup` ma, proprio per questo motivo, questa naming convention è disponibile anche per la classe di inizializzazione, come è illustrato nell'esempio seguente.

Esempio 22.6

```
public class StartupDevelopment
{
```

```
    public StartupDevelopment(IConfiguration configuration)
    {
      // invocato quando l´ambiente è Development...
    }
  }
  public class Startup
  {
    public Startup(IConfiguration configuration)
    {
      // invocato quando l´ambiente non è Development...
    }
  }
```

Se si decide di creare classi `Startup` per ogni environment, come viene mostrato nell'Esempio 22.6, però, è bene andare a modificare il metodo `CreateWebHostBuilder` della classe `Program` perché questo fa un uso tipizzato della classe `Startup` che va cambiato per renderlo generico e specifico dell'ambiente di esecuzione, come dimostrato nell'Esempio 22.7.

Esempio 22.7

```
  public class Program
  {
    public static void Main(string[] args)
    {
      CreateWebHostBuilder(args).Build().Run();
    }
    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
      WebHost.CreateDefaultBuilder(args)
              .UseStartup(Assembly.GetExecutingAssembly().FullName);
            //.UseStartup<Startup>();
  }
```

Nell'Esempio 22.7 è messo in evidenza che non è più possibile indicare in modo esplicito il nome della classe `Startup` perché deve essere calcolato secondo il valore della variabile d'ambiente. Pertanto è necessario fare uso di un override del metodo `UseStartup`, che permette di specificare l'assembly in cui si troverà, quindi, il motore di bootstrapping ASP.NET Core farà il resto. Volendo, è possibile testare il funzionamento andando a impostare il parametro di environment direttamente nella creazione dell'host, come mostrato nell'Esempio 22.8.

Esempio 22.8

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
  WebHost.CreateDefaultBuilder(args)
          .UseEnvironment("Development")
          .UseStartup(Assembly.GetExecutingAssembly().FullName);
```

Come abbiamo già avuto modo di vedere all'interno del Capitolo 3, tutta la configurazione specifica per ogni ambiente viene prelevata dai file `appsettings.{environment-name}.json` e può essere letta all'interno dell'applicazione sia tramite `IConfiguration` sia in maniera fortemente tipizzata. Questo, unito a quanto visto in questo stesso capitolo per le variabili d'ambiente, però, non garantisce alcun tipo di protezione dei dati: infatti, tutte le chiavi, come per esempio connection string per il database ed eventuali password, sono esposte in chiaro e sono leggibili da chiunque. Pertanto è necessario un sistema basato a secret che assicuri un minimo di sicurezza, come abbiamo già visto nei capitoli precedenti.

Una volta definiti gli ambienti di esecuzione, parametrizzato l'applicazione per poter lavorare con environment diversi e aggiunto il supporto alle secret e ad Azure Key Vault, per poter lavorare in sicurezza con dati che non devono essere esposti all'esterno o vulnerabili tramite la macchina stessa, è il momento di pensare al deployment, ovvero al rilascio in un ambiente dedicato dell'applicazione web creata.

Pubblicazione e hosting

Durante l'introduzione di questo capitolo abbiamo illustrato come e quanto sono cambiate le pratiche per il rilascio delle applicazioni per via di diversi fattori quali il tempo, le competenze tecniche e la tipologia di framework. Nonostante tutti questi cambiamenti, però, il flusso in linea generale non cambia e si mantiene su tre aspetti:

- Pubblicazione: il codice sorgente deve essere compilato e distribuito in una cartella del server.

- Scelta del process manager: la scelta è sempre stata implicita su Windows, con IIS per ASP.NET, ma con ASP.NET Core c'è bisogno di un process manager anche per gli ambienti macOS e Linux che sia in grado di riavviare l'applicazione in caso di malfunzionamenti e che riesca a gestire le richieste in ingresso, rimandandole all'applicazione stessa.

- Configurazione del reverse proxy: è una scelta opzionale, da fare in quei casi in cui non si vuole esporre il servizio in modo diretto; così facendo, il reverse proxy prenderà le richieste e le rimanderà al web server in un secondo momento.

La pubblicazione è già stata analizzata nel Capitolo 2 e, sostanzialmente, permette, tramite il comando `dotnet publish` della CLI, di generare DLL, file e dipendenze in una cartella specifica per la pubblicazione, in maniera tale che non venga copiato in alcun modo il codice sorgente originale. Abbiamo anche già visto le differenze tra un deployment di tipo self-contained, in cui nella cartella di pubblicazione verrà aggiunto il runtime, creando così un pacchetto di pubblicazione più grande, rispetto alla pubblicazione classica di tipo framework-dependent, in cui si dà per scontato che il runtime sia già installato fisicamente sulla macchina server di destinazione, e pertanto non discuteremo di ulteriori dettagli.

La scelta del web server, invece, è importante, non solo perché non né si può fare a meno, dato che il sistema in-process serve a rimandare le richieste HTTP verso `HttpContext` all'interno dell'applicazione, ma anche perché prima di decidere quale web server utilizzare, è necessario capire il sistema operativo di destinazione, ed è una scelta strategica. Poiché ASP.NET Core è in grado di funzionare cross-platform, Microsoft ha deciso di rilasciare due implementazioni:

- Kestrel: di default, funziona anche cross-platform.

- HTTP.sys: chiamato in passato WebListener, è l'implementazione esclusiva per Windows, basata sul kernel driver HTTP.sys e HTTP server API.

Abbiamo già descritto i vantaggi con una tabella comparativa a inizio libro nell'uso di uno, dell'altro, o di un server personalizzato costruito sulla base dell'interfaccia `IServer` ma, in generale, poiché Kestrel è di default e il funzionamento è cross-platform, sarà quello che utilizzeremo in futuro.

L'uso di un reverse proxy, inoltre, non è strettamente necessario, poiché Kestrel è considerato, dalla versione 2 di ASP.NET Core production-ready ma ci sono decine di casi in cui ha comunque senso utilizzarlo: limitare l'esposizione dell'host, aggiungere un nuovo strato di sicurezza, limitare il numero di chiamate e il traffico generato e semplificare il load balancing sono solo alcuni esempi. Proprio per queste motivazioni, vedremo ora come configurare NGINX e Apache come reverse proxy per ambienti Linux e macOS.

Configurazione di NGINX

NGINX è un web server molto leggero, che garantisce ottime prestazioni per via del suo approccio asincrono basato su eventi generati dalle richieste HTTP che arrivano in ingresso ed è ottimo per funzionare come reverse proxy con Kestrel. Inoltre, funziona in ambienti Unix/Linux, BSD (macOS) e Windows, ed è perciò in grado di fornire quella componente cross-platform che abbiamo ricercato continuamente all'interno del libro. Per vedere un aspetto diverso e per mantenere una certa semplicità, le demo successive saranno costruite secondo un ambiente macOS con una sola istanza di NGINX non replicata, ma tutto il codice e le configurazioni che vedremo saranno portabili con pochi cambiamenti all'interno degli altri sistemi operativi e delle eventuali repliche.

Considerando che, una volta configurato il reverse proxy, le richieste passeranno prima da NGINX e poi verranno rimandate all'applicazione ASP.NET Core, è necessario configurare opportunamente gli header di forward, altrimenti, qualora nelle applicazioni web ci siano middleware o logiche che leggono questi valori, come per esempio sistemi di tracing delle request, questi potrebbero non funzionare più correttamente. Questo è particolarmente utile anche in scenari in cui si vuole avere un trace completo delle richieste per capire, potenzialmente, che cosa non stia funzionando a livello di routing.

Esempio 22.9

```
public void Configure (IApplicationBuilder app)
{
  app.UseForwardedHeaders(new ForwardedHeadersOptions
  {
    ForwardedHeaders = ForwardedHeaders.XForwardedFor |
      ForwardedHeaders.XForwardedProto;
  });
  app.UseMvc();
}
```

L'Esempio 22.9 mette in evidenza come tramite il metodo `Configure` si possa agire sulla configurazione di `UseForwardedHeaders` per specificare di rimandare gli header `X-Forwarded-For` (che conterrà l'indirizzo IP di origine della richiesta) e `X-Forwarded-Proto` (che conterrà invece il protocollo HTTP o HTTPS).

Una volta che abbiamo preparato l'applicazione a ricevere le richieste, arriva il momento di installare e configurare il web server: per l'installazione si può fare uso del package manager HomeBrew su macOS, con il comando `brew install nginx`, piuttosto che di APT per Linux con il comando `apt-get install nginx`. Per testare il funzionamento, è necessario avviare il web server tramite il comando `sudo nginx start` e, se questo è stato installato correttamente, partirà con la sua configurazione di default, mostrando una pagina HTML statica al raggiungimento dell'indirizzo `http://localhost:8080`. Una volta verificato il corretto funzionamento del servizio out-of-the-box, è necessario istruirlo per lavorare come proxy e quindi rimandare le chiamate. Per farlo, sarà sufficiente modificare la configurazione di default creata da NGINX nel file/etc/nginx/nginx.conf, come mostrato nell'Esempio 22.10.

Esempio 22.10

```
worker_processes 1;
events {
   worker_connections 1024;
}
http {
   include                  mime.types;
   default_type             application/octet-stream;
```

```
    keepalive_timeout        65;
    server {
       listen                80;
       server_name           localhost;
     location / {
       proxy_pass            http://localhost:5000;
       proxy_http_version    1.1;
       proxy_set_header      Upgrade $http_upgrade;
       proxy_set_header      Connection keep-alive;
       proxy_set_header      Host $host;
       proxy_cache_bypass    $http_upgrade;
     }
   }
 }
```

La maggior parte della configurazione esposta dall'Esempio 22.10 non è cambiata rispetto a quanto è previsto di default dal server. I cambiamenti si trovano solamente nella configurazione della porta 80 anziché 8080 e nella configurazione di tutti gli attributi proxy necessari a rimandare la chiamata all'applicazione. In particolare, poiché il server coincide con la macchina stessa, è stato aggiunto localhost, ma nulla vieta, una volta configurato SSL, di aggiungere il proprio dominio "mydomain. com". Le richieste verranno poi passate all'applicazione ASP.NET Core tramite `proxy_pass` che è stato impostato sempre su localhost nella porta 5000 (default di ASP.NET Core). Una volta salvata la configurazione appena impostata, è necessario rilanciare il web server con il comando `sudo nginx restart` e avviare l'applicazione web con il comando `dotnet nome.dll`: se il tutto è stato rappresentato correttamente, l'applicazione sarà raggiungibile sia tramite `http://localhost:5000`, ovvero tramite Kestrel, sia tramite `http://localhost`, `ovvero in reverse proxy da NGINX`.

Nonostante il sistema sia funzionante, a tutti gli effetti c'è ancora un'operazione che è stata eseguita manualmente, ovvero l'avvio dell'applicazione web; pertanto, non si stanno sfruttando i vantaggi di riavvio automatico visti in precedenza per i process manager. Per farlo, bisogna fare uso di launchd (o systemd su Linux) per creare e registrare servizi. Ogni servizio, in ambiente macOS, è rappresentato da un file con estensione `.plist`, in cui sono specificate tutte le proprietà relative a quello che vogliamo venga fatto in completa autonomia, come illustrato nell'Esempio 22.11.

Esempio 22.11

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>KeepAlive</key>
    <true/>
    <key>Label</key>
    <string>Capitolo22</string>
    <key>ProgramArguments</key>
    <array>
          <string>/usr/local/share/dotnet/dotnet</string>
          <string>Capitolo22.dll</string>
    </array>
    <key>RunAtLoad</key>
    <true/>
    <key>StandardErrorPath</key>
    <string>/tmp/dotnet-api-sterr.log</string>
    <key>StandardOutPath</key>
    <string>/tmp/dotnet-api-stdout.log</string>
    <key>UserName</key>
    <string>root</string>
    <key>WorkingDirectory</key>
    <string>/Users/aspitalia/Desktop/Capitolo22/bin/Release/netcoreapp2.1/
publish</string>
    <key>EnvironmentVariables</key>
    <dict>
          <key>ASPNETCORE_ENVIRONMENT</key>
          <string>Production</string>
    </dict>
</dict>
</plist>
```

Il file mostrato nell'Esempio 22.11 viene salvato all'interno del percorso/Library/LaunchDaemons, così che, una volta registrato il servizio, al riavvio della macchina stessa venga avviato in automatico. All'interno di questo file vengono descritti: il nome del servizio con l'attributo `Label`, qual è il comando che deve essere avviato con i suoi parametri (ovvero `dotnet Capitolo22.dll`) con l'attributo `ProgramArguments`, i percorsi per i log di stdout e stderr, la working directory (che corrisponde al percorso in cui ci sarà la DLL che deve essere eseguita, le variabili d'ambiente e, infine, lo username che può avviare il servizio (`root` è l'unica opzione disponibile in LaunchDaemons). Per avviare il servizio, è necessario lanciare in sequenza i comandi mostrati nell'Esempio 22.12.

Esempio 22.12

```
cd /Library/LaunchDaemons
sudo launchctl load -w Capitolo22.plist
sudo launchctl start Capitolo22
```

Una volta registrato il servizio, si può controllare che sia effettivamente partito lanciando il comando `sudo launchctl list`, come mostrato nella Figura 22.4.

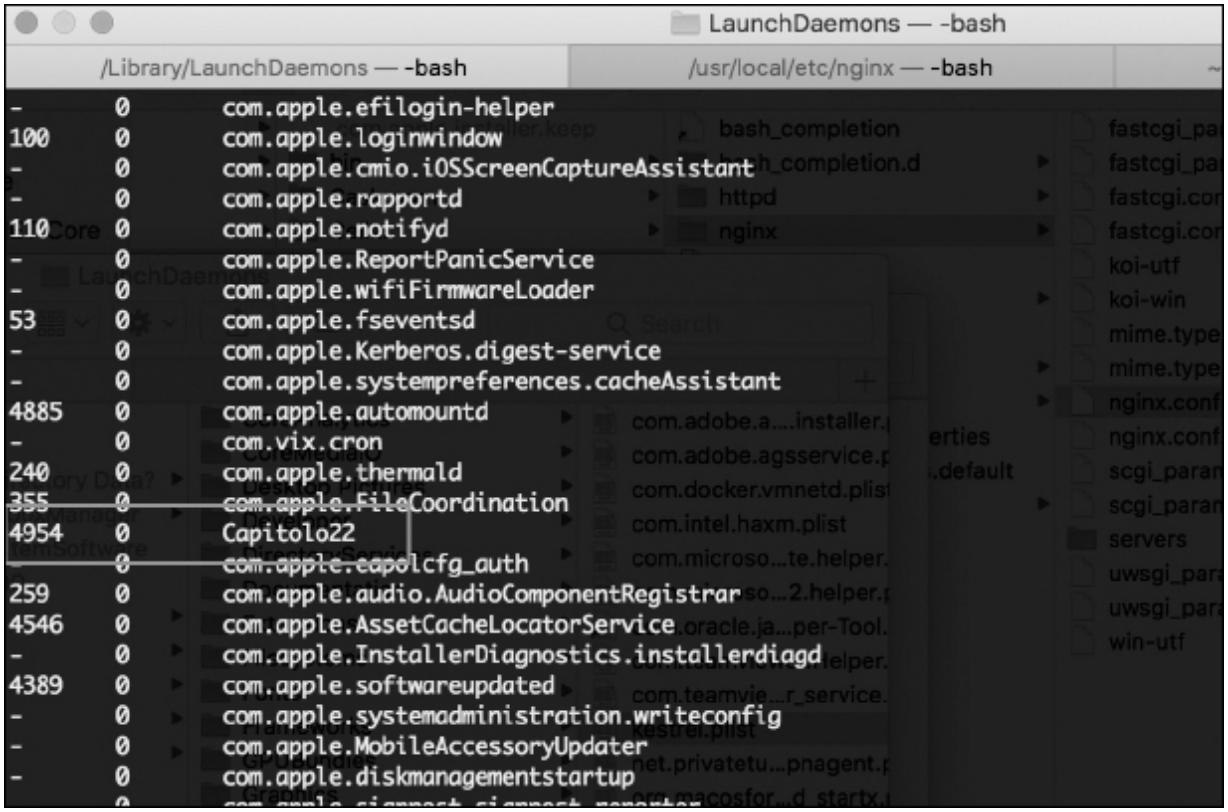


Figura 22.4 – Elenco dei servizi registrati come LaunchDaemons in macOS.

Qualora il servizio sia stato registrato correttamente, come mostrato nella Figura 22.4, con un PID associato, allora la configurazione di NGINX come reverse proxy sarà completata, totalmente automatizzata e resiliente in caso di errori o crash applicativi. Per testimoniare il corretto funzionamento, è sufficiente navigare all'indirizzo `http://localhost` e

dovremmo vedere nuovamente il sito web ASP.NET Core, come illustrato nella Figura 22.5.

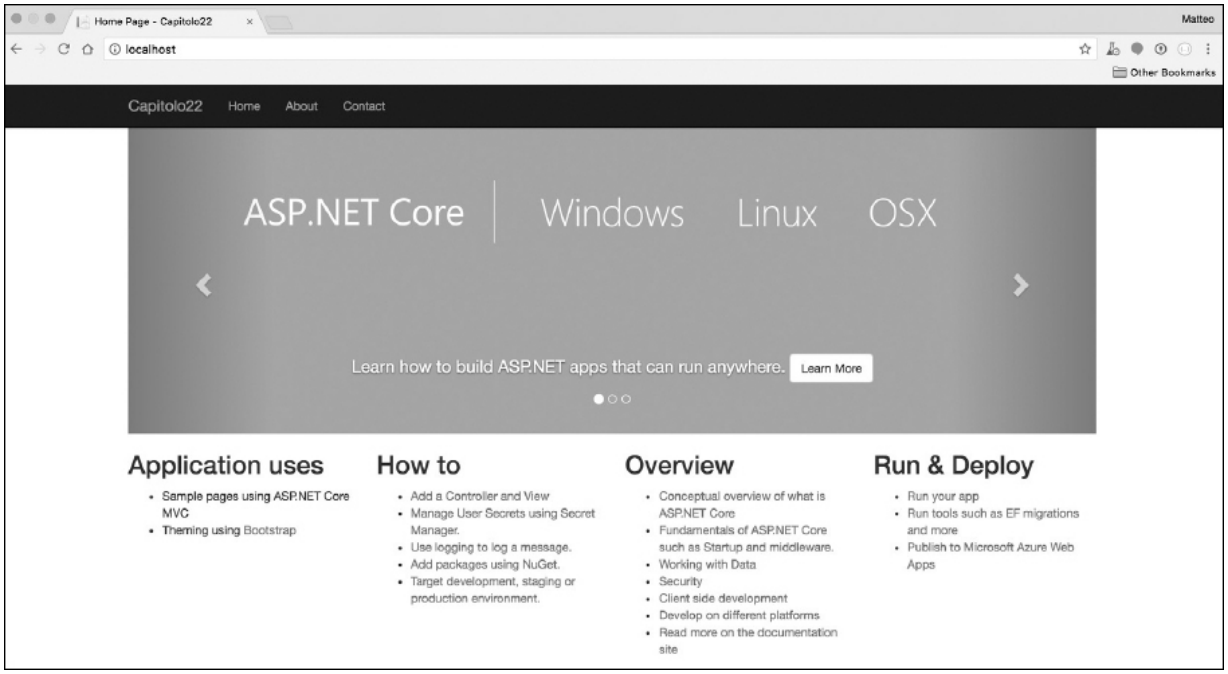


Figura 22.5 – Home page di un'applicazione web ASP.NET Core esposta in reverse proxy tramite NGINX e il servizio di restart automatico.

Poiché l'applicazione viene avviata in autonomia, non ci sarà a disposizione la console per mostrare i log ma, per via del fatto che sono stati esplicitati nel file del servizio `Capitolo22.plist`, sarà sufficiente lanciare il comando `tail -f /tmp/dotnet-api-stdout.log` per vedere lo stream dello standard output e `tail -f /tmp/dotnet-api-stderr.log` per vedere gli eventuali errori generati.

Nella configurazione di base documentata nell'Esempio 22.10, però, manca tutta la parte relativa alla sicurezza stessa del server, e pertanto è necessario andare ad aggiungere sul nodo alcuni header fondamentali, come illustrato nell'esempio seguente.

Esempio 22.13

```
server {
        listen *:443 ssl;
```

```
        add_header Strict-Transport-Security "max-age=63072000";
        add_header X-Frame-Options DENY;
        add_header X-Content-Type-Options nosniff;
}
```

Nell'Esempio 22.13 si può notare come siano stati aggiunti tre header:

- HTTP Strict-Transport-Security: HSTS è attivo di default in tutti i progetti a partire da ASP.NET Core 2.1 e permette di gestire tutte le richieste inviate dai client su HTTPS; per questo va accompagnato da certificati SSL e limiti di durata tramite `max-age`.

- X-Frame-Options: se impostato con valore `DENY` garantisce protezione dagli attacchi di clickjacking, in quanto non permette l'uso di frame con link a fonti esterne.

- X-Content-Type-Options: se impostato con valore `nosniff`, garantisce protezione dagli attacchi di MIME-Type sniffing, in cui il browser potrebbe cambiare l'header Content-Type e interpretare un oggetto di un tipo come di un altro.

La configurazione del server, poiché in questo caso risponderà sulla porta 443 anziché sulla porta 80, come abbiamo visto in precedenza, può essere affiancata alla configurazione mostrata nell'Esempio 22.10 e il sistema risponderà in reverse proxy sempre in modo adeguato, rimandando le richieste su HTTPS. Dal momento che il resto della configurazione del server dipende principalmente dalle proprie esigenze di produzione, non affronteremo temi più avanzati nel corso di questo capitolo ma rimandiamo alla documentazione ufficiale disponibile su `http://aspit.co/bot` e proseguiamo vedendo un web server alternativo, come Apache.

Configurazione di Apache HTTP Server

Apache HTTP Server (più comunemente Apache o httpd) rappresenta il secondo web server che si può sfruttare per applicare la configurazione di reverse proxy verso Kestrel, dato che anche questo è disponibile su più piattaforme, tra cui Linux, Windows e macOS. A oggi è tra i web

server più utilizzati, probabilmente per via del fatto che il suo sviluppo è completamente open-source attraverso la Apache Software Foundation e che il primo rilascio fu effettuato nell'ormai lontano 1995. A livello puramente architetturale, differisce da NGINX per via del fatto che è completamente modulare: ogni componente è indipendente ed è in grado di gestire una singola funzionalità ma tutto il coordinamento tra i vari moduli avviene tramite il core, che prende in ingresso le richieste e le gira in sequenza a tutti i moduli che sono stati registrati. Poiché il core, per funzionare, ha bisogno continuamente di avere accesso a tutti i moduli, è necessario avere un servizio che tramite polling interroghi continuamente le singole componenti per aggiornare il loro stato, e per questo è meno performante rispetto a NGINX. A livello concettuale, quanto visto per NGINX non cambia rispetto a quanto vedremo con Apache: una volta effettuata l'installazione del web server, sarà necessario andare a cambiare la configurazione secondo le proprie esigenze e registrare lo stesso servizio già visto nell'Esempio 22.11 per ottenere lo stesso risultato. Anche quanto visto sulla configurazione e sulla pubblicazione dell'applicazione web stessa non cambia, e pertanto rimane consigliato abilitare il forwarding degli header, come è illustrato nell'Esempio 22.9.

L'installazione del server Apache dipende dal sistema operativo che si ha a disposizione ma, per coerenza, continueremo a utilizzare macOS per gli esempi a seguire. Pertanto si può fare uso del package manager HomeBrew e lanciare il comando `brew install httpd` (o `yum install httpd mod_ssl`) per procedere all'installazione nel caso in cui non sia già disponibile nella macchina server. Per verificare che il tutto sia stato configurato correttamente, si può lanciare il comando `sudo apachectl start` e aprire il browser in `http://localhost:8080` per vedere la prima pagina, come è illustrato nella Figura 22.6.

Verificato che il server è in grado di partire, esattamente com'è stato fatto in precedenza per NGINX, è necessario configurare il web server per il reverse proxy. In questo caso, poiché il sistema è composto da moduli, bisogna andare a modificare un modulo chiamato `httpd-vhosts.conf` contenuto nella cartella extra a partire dal percorso in cui è stato installato il web server (per esempio: /etc/local/ httpd) come è evidenziato nell'Esempio 22.14.

Figura 22.6 – La home page di Apache è una pagina `index.html` disponibile in un percorso variabile in base all'installazione effettuata (su macOS di default è /etc/local/var/www/ ma può essere cambiata nella configurazione di httpd) e pertanto può essere personalizzata.

Esempio 22.14

```
<VirtualHost *:8080>
    ProxyPreserveHost On
    ProxyPass / http://127.0.0.1:5000/
    ProxyPassReverse / http://127.0.0.1:5000/
    ServerName http://127.0.0.1
    ServerAlias localhost
    ErrorLog /tmp/error.log
    CustomLog /tmp/info.log common
</VirtualHost>
```

Come si può notare, nell'Esempio 22.14 sono stati registrati per il proxy gli indirizzi verso l'applicazione ASP.NET Core, che rimane in ascolto sulla porta 5000 (per HTTP), quindi, quando verrà fatta di nuovo la chiamata a `http://localhost:8080`, verremo reindirizzati all'applicazione e non più alla pagina di benvenuto vista nella Figura 22.7. Purtroppo, al momento, l'applicazione non è ancora visibile perché mancano due dettagli fondamentali: il primo è configurare il tutto in modo che il modulo httpd-vshosts venga caricato, il secondo invece è riavviare httpd con la nuova configurazione e il nuovo modulo. Per comunicare al core che il modulo `httpd-vhosts` deve essere caricato, bisogna andare a modificare la configurazione contenuta nel file `httpd.conf` decommentando le righe di codice mostrate nell'Esempio 22.15.

Esempio 22.15

```
LoadModule proxy_html module lib/httpd/modules/mod_proxy_html.so
LoadModule proxy_module lib/httpd/modules/mod_proxy.so
Include /usr/local/etc/httpd/extra/httpd-vhosts.conf
```

Le righe mostrate nell'Esempio 22.15 istruiscono il core su quale configurazione debba essere caricata ma, poiché all'interno dell'host vengono richiesti attributi relativi ai proxy per rimandare le chiamate, devono essere caricati tutti i moduli associati. A questo punto è sufficiente riavviare il web server con il comando `sudo httpd restart` per fare in modo che prenda la nuova configurazione per core e tutti i suoi componenti aggiuntivi per vedere a tutti gli effetti l'index dell'applicazione web ASP.NET Core.

Per quanto riguarda il resto della configurazione, valgono le stesse considerazioni già fatte per NGINX: se si vuole che il processo parta in automatico e non ci sia più il bisogno di eseguire a mano il comando `dotnet run`, allora si può registrare lo stesso servizio già discusso nell'Esempio 22.11 mentre, se si preferisce avere l'applicazione con HTTPS, sono comunque obbligatori i certificati SSL e l'apertura della porta 443 e, infine, se si vogliono prevenire gli attacchi di sicurezza come clickjacking e MIME-type sniffing, sono ancora necessarie le configurazioni a livello di header che possono essere fatte direttamente nel file `httpd.conf`. Per questi e altri scenari più avanzati, rimandiamo alla documentazione ufficiale su: `http://aspit.co/bou`.

E con l'approfondimento di Apache si conclude la panoramica su come possono essere configurati due web server che, a differenza di IIS, sono in grado di funzionare su piattaforme diverse da Windows. Per chi non ha queste esigenze, affronteremo ora il tema della distribuzione su IIS.

Configurazione di IIS

Internet Information Service (IIS) è sicuramente il web server più conosciuto per chi proviene dal mondo Windows e rimane uno dei web server più sicuri al mondo. La sua complessità è notevole ma, all'incirca come già visto per Apache, la sua architettura è formata da diversi moduli componibili fra di loro. Al contrario di quanto abbiamo già visto in precedenza con Apache e NGINX, in questa parte del capitolo daremo per scontata l'installazione e la configurazione stessa di IIS poiché

dipende dalla versione di Windows (sulla parte server, per esempio, è attiva di default, mentre sulla parte client va attivata tramite le funzionalità aggiuntive) e dai vari ruoli che si vogliono aggiungere, come Windows Authentication o WebSockets. Nonostante l'installazione sia già data per scontata, questo ancora non consente l'esecuzione delle applicazioni ASP.NET Core out-of-the-box perché .NET Core è un framework relativamente nuovo e IIS non ha le informazioni necessarie per sapere come comportarsi: per questo è necessario procedere all'installazione del **.NET Core Windows Server Hosting bundle**, ovvero di un pacchetto che contiene runtime e librerie relative al framework, in aggiunta al modulo AspNetCoreModule di IIS, che è proprio la componente che sa come comunicare con l'applicazione. Il download di questo bundle è possibile all'url: `http://aspit.co/boy`. Una volta installato e riavviata la macchina, sarà visibile all'interno dei moduli utilizzabili dal web server, come mostrato nella Figura 22.7.

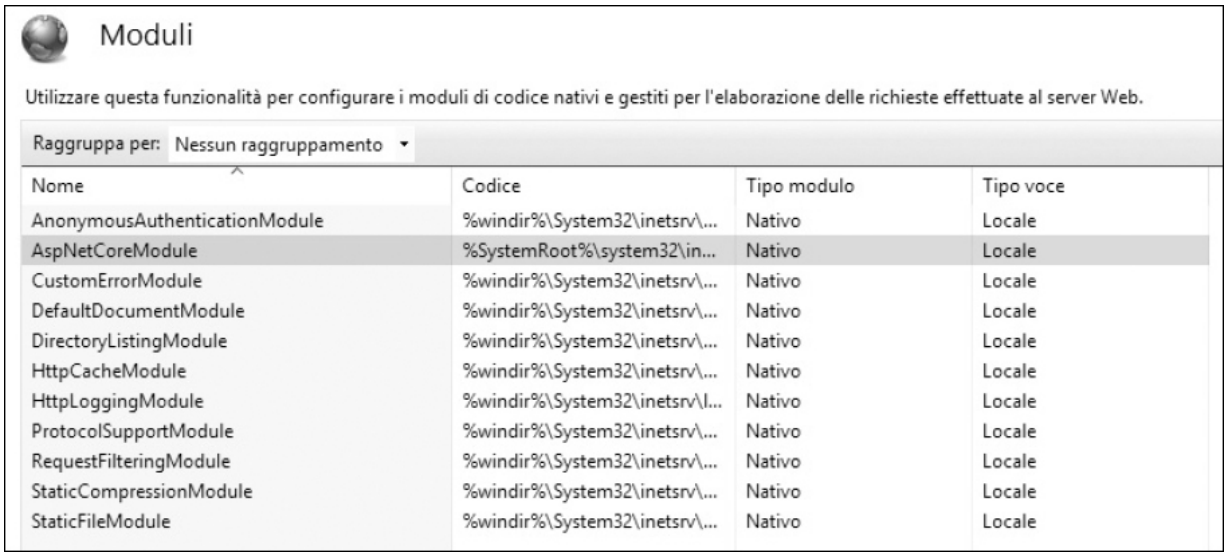


Figura 22.7 – Il modulo `AspNetCoreModule` è visibile solamente quando è stato installato correttamente il .NET Core Windows Server Hosting bundle e il runtime di ASP.NET Core.

La parte importante è che l'AspNetCoreModule non solo è in grado di eseguire le applicazioni ASP.NET Core ma permette anche la comunicazione in reverse proxy con Kestrel, senza dover cambiare una

riga di codice dell'applicazione poiché il tutto viene già gestito da `IWebHostBuilder` della classe `Startup`. Inoltre, dal momento che rimanda tutto il traffico verso Kestrel, questo modulo ci permette anche di gestire scenari più avanzati relativi a sicurezza, logging e configurazione applicativa e del server stesso.

Una volta completato il setup di base, arriva il momento di creare la configurazione applicativa: ogni applicazione per girare ha bisogno di un suo application pool, ovvero di un sistema che garantisca isolamento tra le applicazioni presenti sullo stesso server, così che se una genera errori oppure va in crash, non è in grado di danneggiare le altre. Si deve quindi procedere e creare un nuovo application pool, in cui la particolarità, rispetto a quanto si è abituati dal passato con le applicazioni ASP.NET, è che ASP.NET Core non ha bisogno del CLR per partire e pertanto, come dimostra la Figura 22.8, si può impostare sul valore "No managed code".

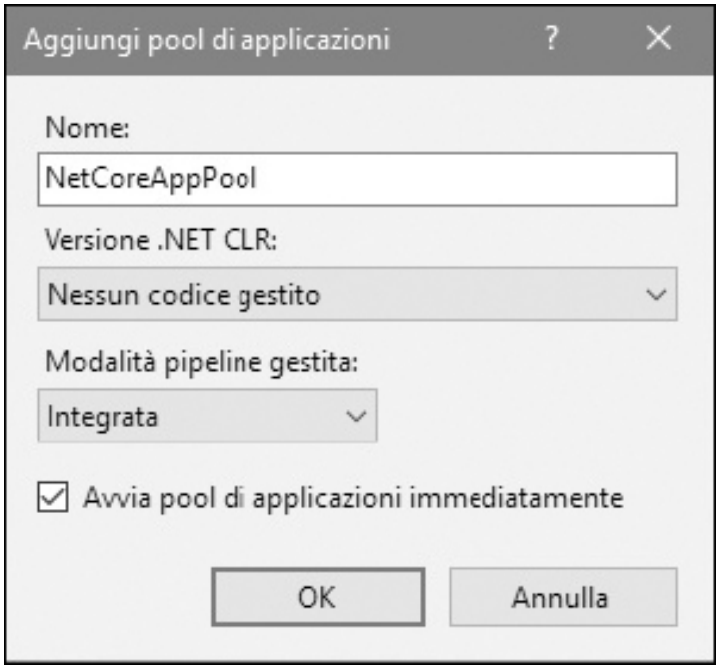


Figura 22.8 – Il nuovo application pool si può configurare con click destro sul nodo degli application pool e va impostato in modalità "No managed code" poiché .NET Core non richiede il Desktop CLR.

Una volta definito l'application pool, è possibile andare ad aggiungere il sito web, cliccando con il tasto destro sul nodo "Sites" e creandone uno

nuovo. Alcuni dei parametri sono esposti nella Figura 22.9, mostrata qui di seguito.

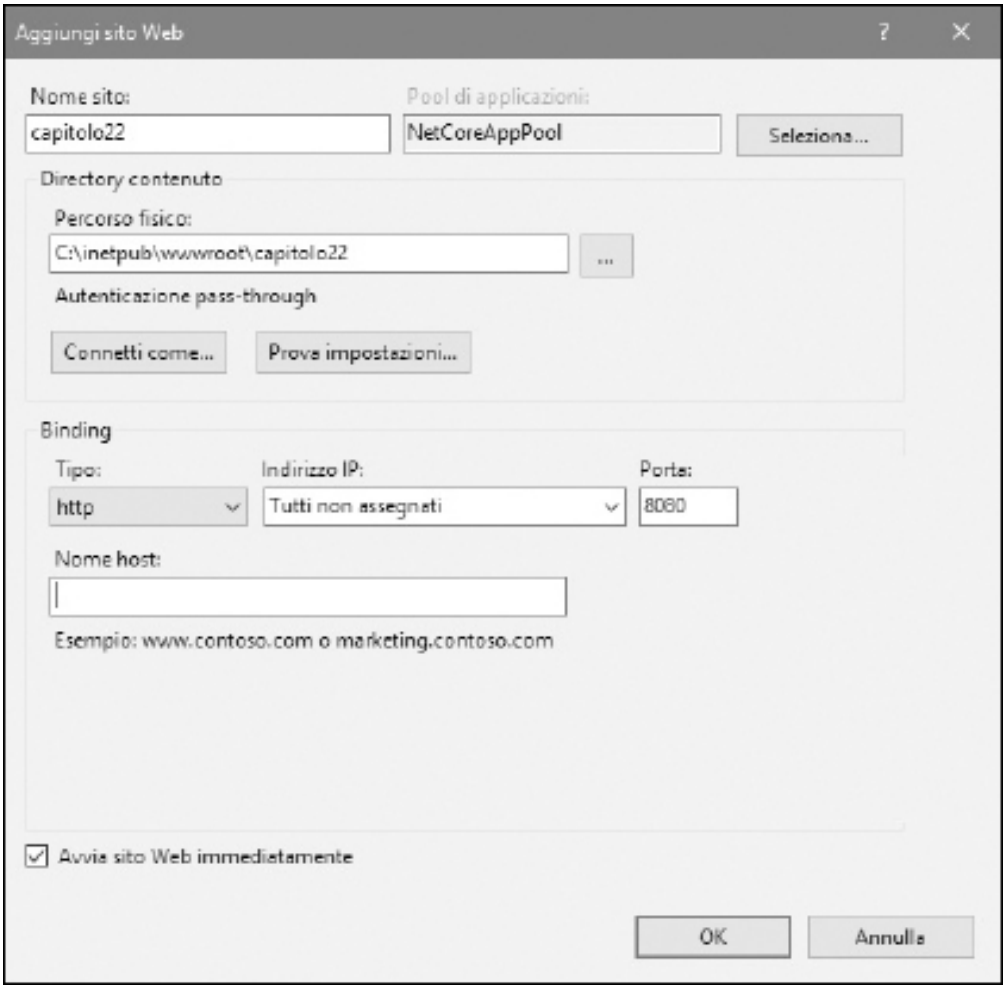


Figura 22.9 – Configurazione di nuovo sito web all'interno di IIS per ospitare l'applicazione ASP.NET Core.

Dalla Figura 22.9 si possono notare alcune caratteristiche: il nome del sito è rilevante solamente a livello dell'IIS, il nome dell'host, al contrario, è rilevante quando lo si vuole esporre all'esterno tramite certificati SSL e domini personalizzati, l'application pool selezionato è quello impostato nel passaggio precedente e, infine, il percorso fisico corrisponde alla cartella in cui è stato lanciato il comando `dotnet publish`. Andando ad analizzare l'output generato dalla pubblicazione di un'applicazione ASP.NET Core, troviamo, tra tutti, un file fondamentale per l'avvio

dell'applicazione, ovvero il web.config mostrato nell'Esempio 22.16, che serve a specificare all'IIS tutti i parametri relativi all'applicazione e a IIS stesso.

Esempio 22.16

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <handlers>
      <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModule"
resourceType="Unspecified" />
    </handlers>
    <aspNetCore processPath="dotnet" arguments=".\Capitolo22.dll"
stdoutLogEnabled="false" stdoutLogFile=".\logs\stdout" />
  </system.webServer>
</configuration>
```

Nell'Esempio 22.16, infatti, è evidente che nel nodo relativo al web server (IIS) si sta specificando che deve essere caricato il modulo `AspNetCoreModule`, proprio quello che è stato installato con il bundle: è grazie a questo che IIS, partendo, sarà in grado di capire che deve caricare il modulo corretto, e quindi avviare i comandi definiti dagli attributi `processPath` e `arguments` del nodo `aspNetCore`, che rappresentano, guarda caso, i comandi di avvio dell'applicazione stessa. La generazione del file `web.config` è automatica sui progetti che hanno impostato come target di progetto `Microsoft.NET.Sdk.Web` e in cui non è già presente un file che sia personalizzato. Questo passaggio è obbligatorio e fondamentale, perché senza di lui l'applicazione stessa non sarebbe in grado di partire e pertanto questo file di configurazione non va mai rimosso. Se tutti i passaggi elencati sono stati eseguiti correttamente, navigando all'indirizzo `http://localhost:8080` (o nella porta definita in fase di setup), saremo nuovamente in grado di visualizzare l'applicazione creata.

La pubblicazione all'interno dei web server che abbiamo affrontato finora è piuttosto semplice ma richiede ancora una cosa che, nel mondo moderno, non è detto che sia più disponibile: il server fisico. Sempre più spesso, infatti, si sta cercando di spostare tutti i workload in ambienti cloud, in modo da risparmiare quanto più possibile in termini di costi e

tempi di rilascio, ma garantire al tempo stesso una maggiore scalabilità sia orizzontale sia verticale, praticamente immediata, di migliaia di istanze che a oggi non è possibile in data center on-premise. Proprio per via di queste premesse, discuteremo ora di com'è possibile portare la stessa applicazione web anche in un ambiente cloud pubblico di Microsoft Azure e ne illustreremo i servizi gestiti.

Pubblicazione con Microsoft Azure

Microsoft Azure offre, come gli altri cloud vendor, la possibilità di creare servizi e scalarli potenzialmente all'infinito, secondo le proprie esigenze, con una novità per quanto riguarda la fatturazione: al contrario di quanto avviene on-premise, in cui il costo dell'hardware e di eventuali servizi necessari alle applicazioni devono essere sostenuti a priori, con l'avvento del cloud i costi sono limitati alle risorse che vengono effettivamente utilizzate. Entrando nel mondo delle applicazioni web, abbiamo già visto come sia possibile configurare un proprio web server e fare l'hosting. Pertanto, se si volesse andare nel cloud, il primo passo sarebbe di utilizzare il servizio delle macchine virtuali. Il problema nell'adottare una soluzione di questo tipo è solitamente relativo ai costi che si riflettono su diversi aspetti: manutenzione del sistema operativo, gestione delle patch, configurazione del web server, configurazione della scalabilità delle macchine virtuali, gestione del cluster per mantenere alta l'affidabilità, creazione del load balancing e così via. Analizzando meglio questi punti, è abbastanza evidente che non si trae alcun vantaggio nell'andare nel cloud con una soluzione del genere, perché ci si porterebbe dietro gli stessi problemi dell'on-premise, meno la gestione dell'hardware, e per questo conviene adottare quelli che vengono definiti servizi gestiti (PaaS), in cui tutto l'hardware e il servizio stesso vengono gestiti direttamente da Microsoft, lasciandoci solamente possibilità di personalizzazioni sull'ambiente ma senza avere l'accesso alle macchine. Il tutto si traduce in una serie di vantaggi e di riduzioni dei costi, sia da parte del cloud, perché sono coinvolte meno risorse, sia da parte delle aziende, perché sono necessarie meno persone specializzate sull'infrastruttura.

Il servizio gestito in Microsoft Azure, che si occupa dell'hosting delle applicazioni web, si chiama App Service e per utilizzarlo è richiesta solamente l'attivazione di una sottoscrizione (anche gratuita) che può essere fatta da questo link: `http://aspit.co/boz`. Una volta attivato l'account, tutto il resto delle operazioni può essere fatto direttamente all'interno di Visual Studio tramite una serie di menù guidati: il primo passo è quello di cliccare con il tasto destro sul progetto che si vuole pubblicare e quindi scegliere l'opzione "Publish". Questo mostrerà a schermo un menù come quello evidenziato nella Figura 22.10.

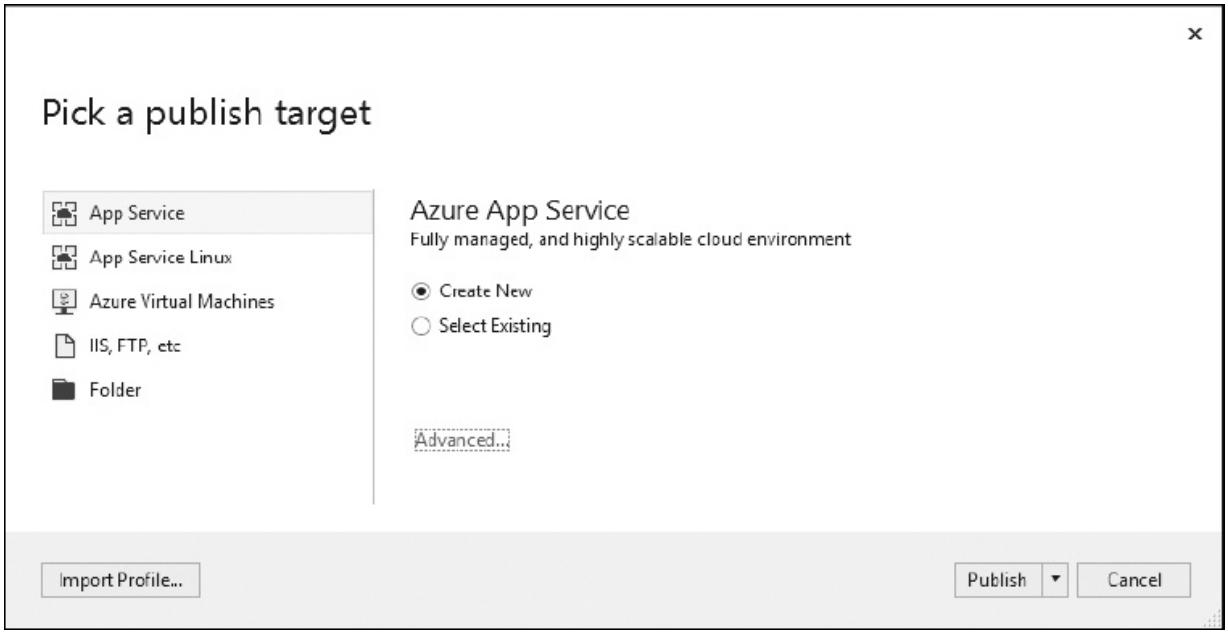


Figura 22.10 – Il menù di pubblicazione di Visual Studio offre diversi target, tra cui gli Azure App Service (Windows o Linux), macchine virtuali, IIS/FTP e cartelle definite in locale.

Come si può notare dalla Figura 22.10, sono disponibili diverse opzioni, tra cui la stessa macchina virtuale in cui si dovrà configurare il web server, piuttosto che la pubblicazione tramite IIS, FTP o una cartella, che non fanno altro che richiamare il comando `dotnet publish` in un percorso specifico. Tra le opzioni relative al cloud, invece, ci sono due livelli di App Service perché, dato che si sta sviluppando un'applicazione ASP.NET Core, si potrebbe voler distribuire l'applicazione non solo su

Windows ma anche su ambiente Linux. Discuteremo successivamente di come questo venga fatto, parlando dei Docker container. Prima di creare il servizio, dato che al momento da portale non è stato ancora creato, è necessario assicurarsi che le impostazioni di pubblicazione dell'applicazione stessa siano corrette. Pertanto, si può cliccare sul pulsante "Advanced" ed effettuare le modifiche, come illustrato nella Figura 22.11.

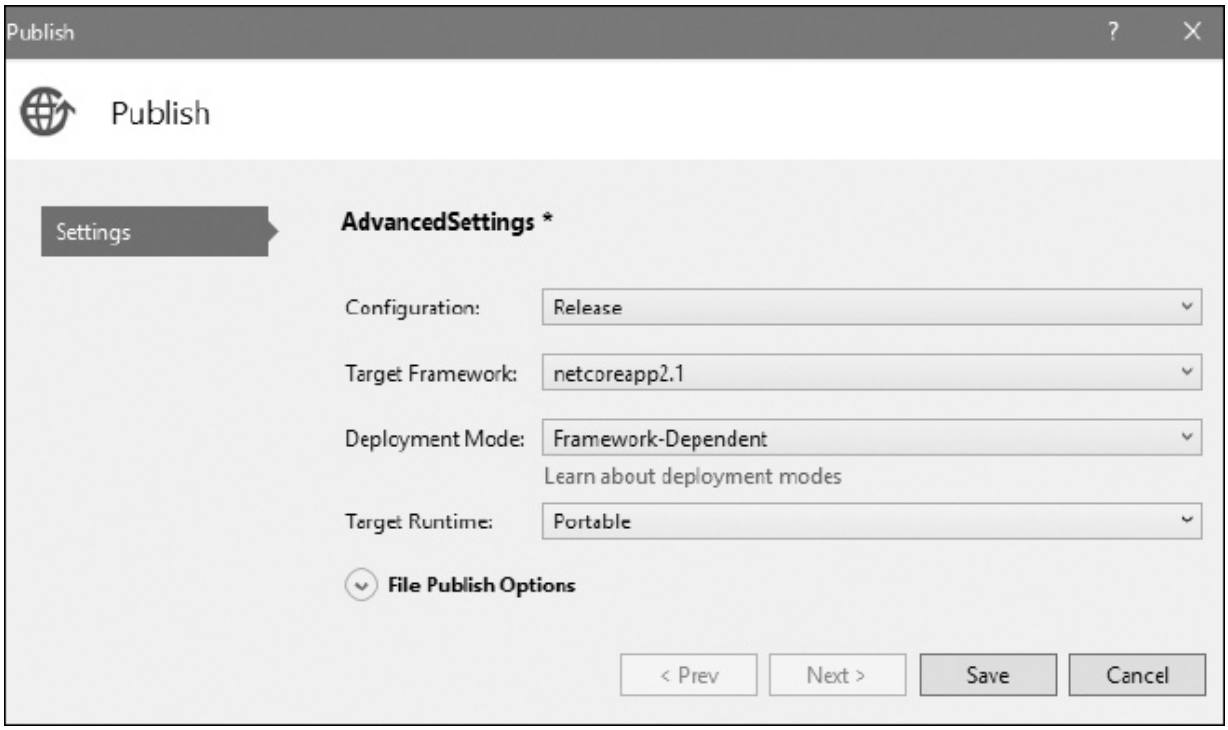


Figura 22.11 – Ogni progetto ASP.NET Core può avere diverse modalità di pubblicazione (framework-dependent o self-contained).

Il servizio degli App Service prevede già un'installazione di .NET Core, perciò si può sfruttare una tipologia di deployment framework-dependent, come mostrato nella Figura 22.11, ma in caso in cui ci sia l'esigenza di un altro runtime, si può comunque fare il deployment di tipo self-contained. Una volta salvate le impostazioni di pubblicazione e cliccato il pulsante "Publish" del menù della Figura 22.10, verrà richiesto di fare il login con l'account Microsoft con il quale è stata creata la

sottoscrizione di Azure e quindi verrà mostrato il menù della Figura 22.12, in cui sarà possibile creare la nuova risorsa.

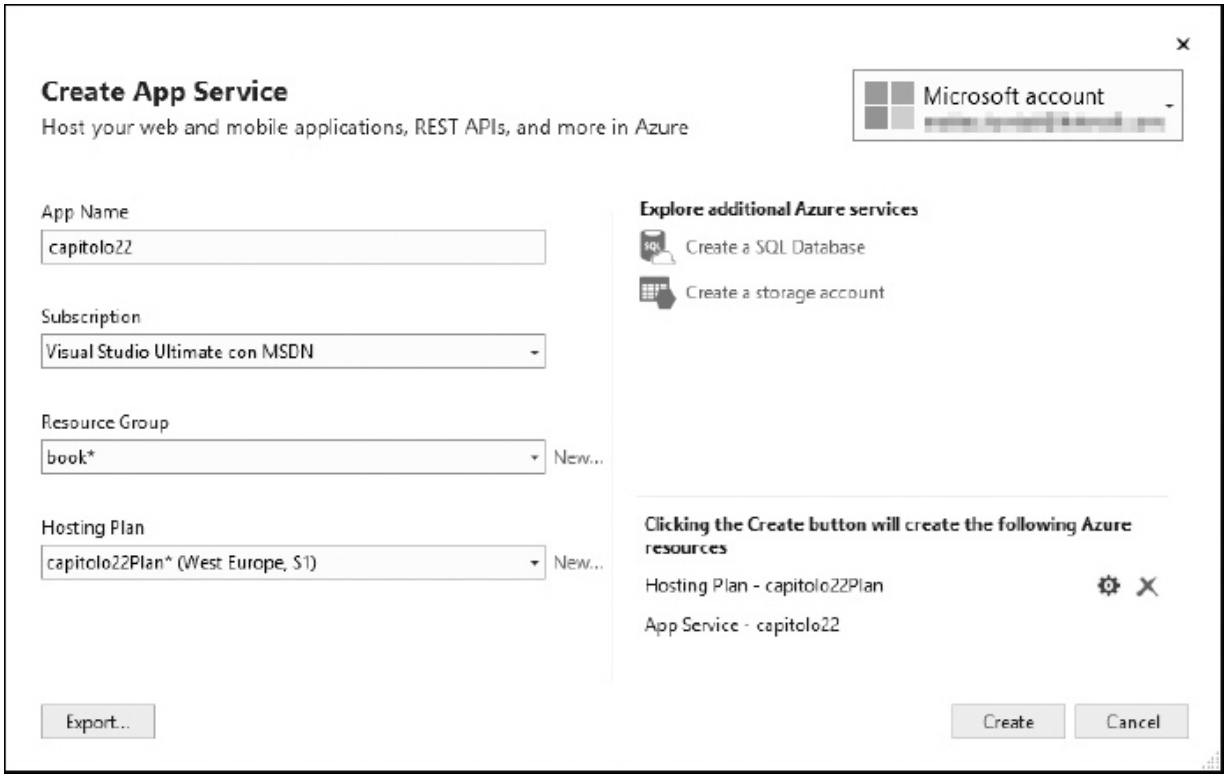


Figura 22.12 – Il menù di creazione di una nuova istanza di Azure App Service permette di aggiungere anche servizi esterni, come un Azure SQL Database o un Azure Storage Account per salvare dati.

La creazione di una nuova risorsa di tipo App Service richiede quattro parametri:

- App Name: rappresenta il nome DNS che avrà l'applicazione, perciò deve essere unico in tutto il mondo. L'URL completo dell'applicazione sarà: `https://{AppName}.azurewebsites.net` di default, ma sarà possibile cambiarlo tramite portale, con l'aggiunta di certificati SSL e domini personalizzati.

- Subscription: la sottoscrizione attiva verso la quale verrà effettuata la fatturazione mensile.

- Resource Group: permette di rappresentare un elenco di risorse che sono correlate fra di loro (come, per esempio, un sito web e il relativo database) in un gruppo in cui sono tutte visibili.

- Hosting plan: indica il piano di servizio, ovvero la grandezza della macchina di una singola istanza che eseguirà l'applicazione e la sua posizione geografica.

La configurazione del piano di servizio viene fatta tramite un menù secondario, come è visibile nella Figura 22.13.

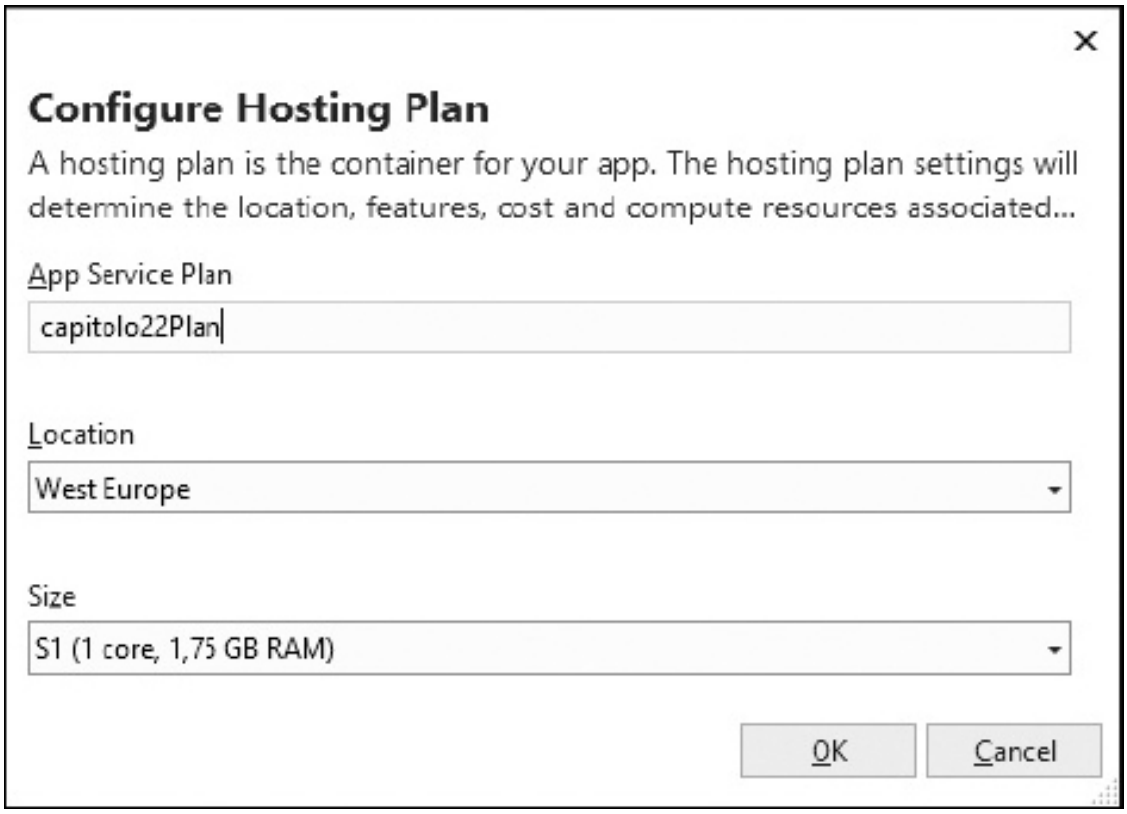


Figura 22.13 – La configurazione del piano di servizio include la scelta del data center e della dimensione della singola istanza dell'App Service.

Una volta configurata la risorsa base per l'hosting dell'applicazione web, è eventualmente possibile configurare il rilascio anche delle risorse collegate come, per esempio, il database. La procedura sarà molto simile, però, su Microsoft Azure, verrà creata, per esempio, una risorsa

gestita di tipo SQL Database, che permetterà di gestire i dati relazionali come se fosse un vero e proprio SQL Server. Cliccando sul pulsante "Create" del menù mostrato nella Figura 22.12, verranno eseguite due operazioni: la prima è la creazione, all'interno della cartella "Properties" del progetto, di un file chiamato `{AppName}-WebDeploy.pubxml` contenente tutta la configurazione appena effettuata, mentre la seconda è la pubblicazione all'interno del servizio cloud. Quando l'operazione di pubblicazione si concluderà, saremo reindirizzati dal browser al vero e proprio sito web appena creato, come risulta visibile nella Figura 22.14.

Le pubblicazioni che verranno eseguite successivamente, per via di modifiche all'interno del codice, saranno molto più rapide perché la configurazione è già salvata all'interno del publishing profile e, per via del meccanismo di deployment scelto, verranno pubblicate solamente le modifiche e non tutta l'applicazione.

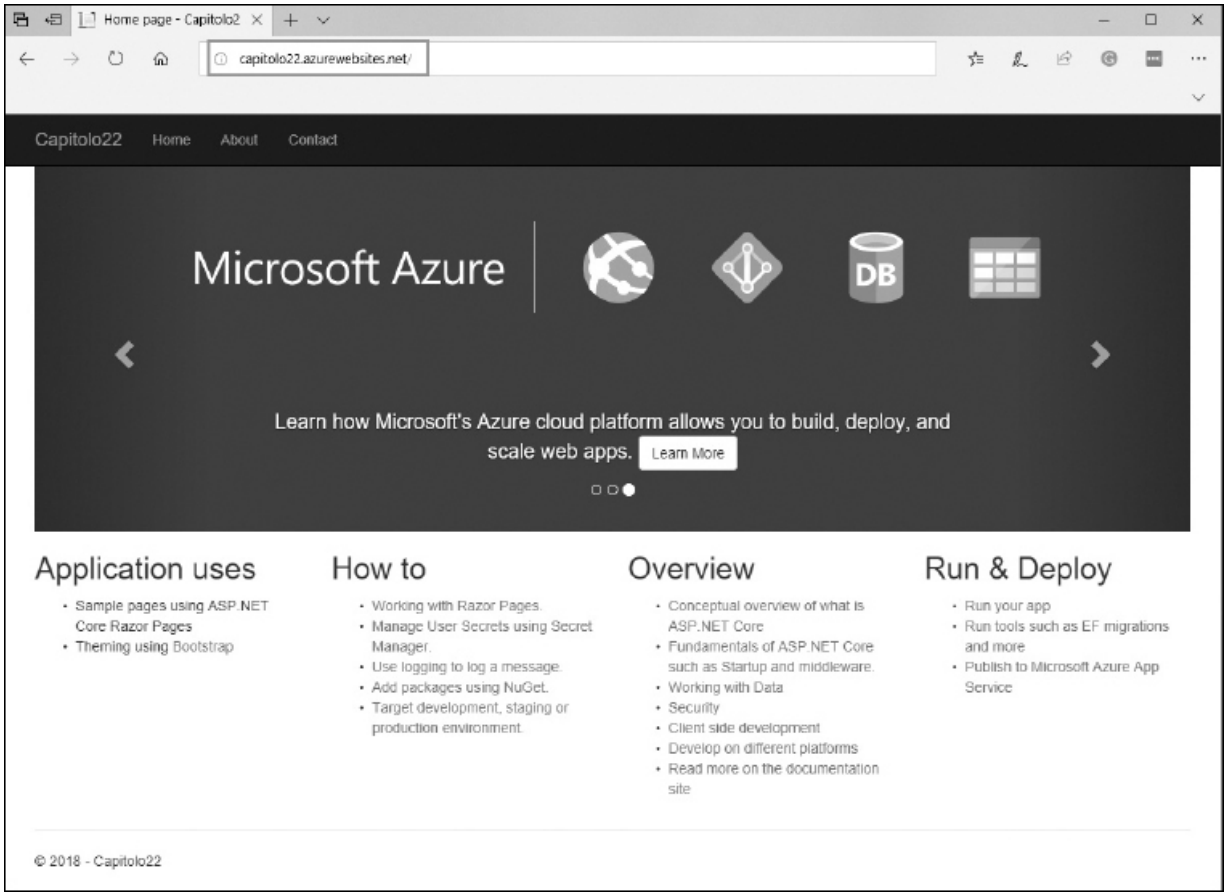


Figura 22.14 – L'applicazione distribuita su App Service è raggiungibile tramite un URL pubblico.

Accedendo al portale su `https://portal.azure.com` sarà possibile modificare alcune delle impostazioni del servizio creato, tra cui, per esempio, l'attivazione delle WebSocket per SignalR, piuttosto che HTTP/2 o, ancora, l'impostazione delle variabili d'ambiente come `ASPNETCORE_ENVIRONMENT` per modificarne la tipologia di esecuzione al volo, come mostrato nella Figura 22.15.

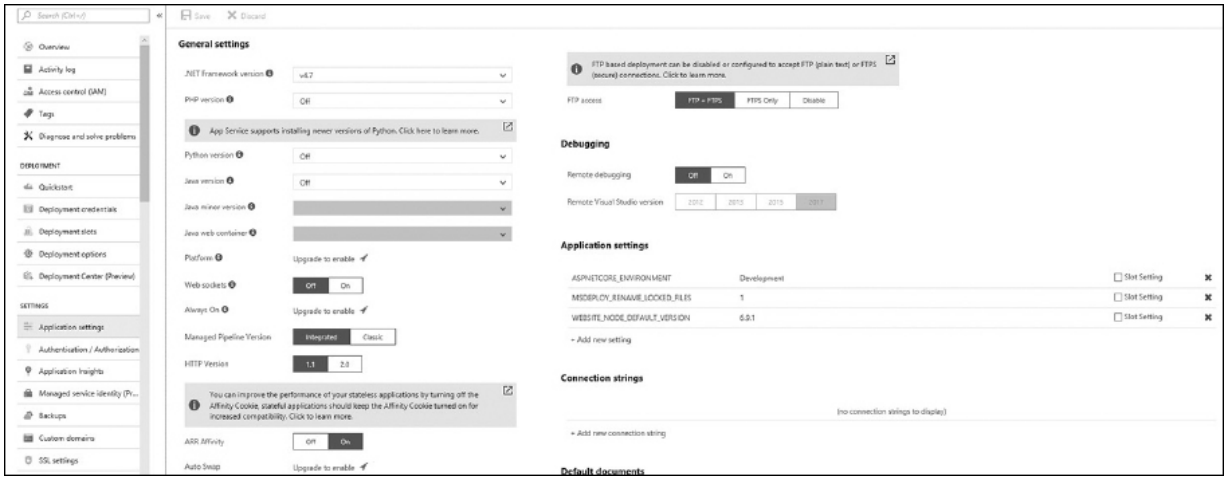


Figura 22.15 – All'interno dell'App Service specificato, si possono cambiare le impostazioni relative all'applicazione stessa ma anche all'ambiente sulla quale viene eseguita.

Un aspetto particolarmente interessante degli App Service è che offrono la possibilità di attaccare il debugger di Visual Studio per determinare, per esempio, come mai alcuni problemi si verificano solamente una volta pubblicata l'applicazione. Questa funzionalità si chiama Remote Debugging e va attivata in modo esplicito, selezionando opportunamente la versione di Visual Studio che si desidera utilizzare direttamente all'interno del portale di Azure, ed è inoltre necessario che l'applicazione distribuita nell'App Service sia stata compilata in modalità di debug. All'interno di Visual Studio, aprendo la finestra del Server Explorer, sarà possibile scegliere tra i nodi Azure -> App Service ->

Resource Group l'applicazione che si deve debuggare e, cliccando con il tasto destro del mouse sopra di essa, selezionare l'opzione "Attach Debugger", come mostrato nella Figura 22.16.

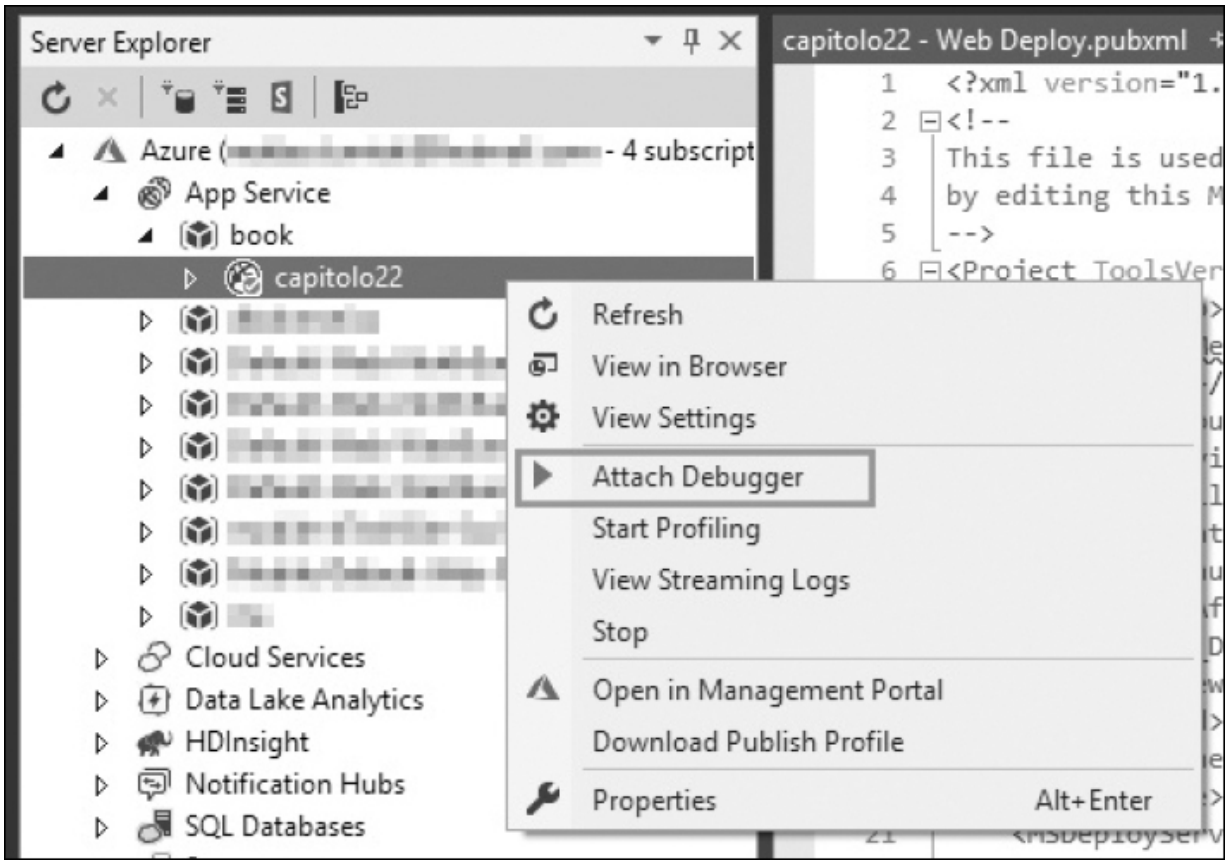


Figura 22.16 – L'opzione "Attach Debugger" è disponibile tramite il menù secondario dal Server Explorer sull'istanza di App Service selezionata.

Questa funzionalità è estremamente potente e non deve essere utilizzata in scenari di produzione: ogni qualvolta si incontri, per esempio, un breakpoint all'interno di una action tramite una route specificata dall'utente, tutta l'esecuzione dell'applicazione verrà bloccata fino a quando non verrà eseguito lo step successivo ed eventualmente rilasciato il debugger. Testare in remoto è comunque molto utile perché, essendo il servizio gestito, non è possibile essere a conoscenza dei dettagli implementativi dei server che offrono il servizio stesso; pertanto, attivando il debugger, si potrebbero scoprire situazioni non verificabili e non riproducibili nell'ambiente di sviluppo o di staging.

Gli Azure App Service includono molte altre funzionalità, fra cui la personalizzazione dei domini, l'autenticazione, la gestione delle performance e logging tramite Application Insights, o la scalabilità automatica ma, poiché non interessano il tema centrale del libro, rimandiamo alla documentazione ufficiale su `http://aspit.co/bo0` per approfondire le potenzialità di questi servizi gestiti.

In generale, soluzioni come quelle che abbiamo affrontato all'interno di questo capitolo funzionano, ma non sono l'ideale nei casi in cui si stia sviluppando un sistema orientato ai micro-servizi: in questi casi specifici, infatti, distribuire e orchestrare tutte le versioni, piuttosto che gestire le dipendenze fra i vari servizi, diventa via via più complesso, pertanto c'è bisogno di un sistema che semplifichi ulteriormente il rilascio.

Pubblicazione con Docker

Uno dei problemi più sentiti da parte degli sviluppatori è che non è possibile lavorare con ambienti completamente isolati e replicabili. Infatti, spesso si sente parlare del fenomeno "it works on my machine" (letteralmente "funziona sul mio PC") per via del fatto che l'installazione di una nuova dipendenza, se non comunicata agli altri membri del team di sviluppo o configurata correttamente, potrebbe portare a una instabilità del sistema fino al non funzionamento dello stesso, cosa che in ambienti di produzione non dovrebbe succedere. Mettendosi invece nei panni del team di operation che deve gestire la messa in produzione delle varie applicazioni, il tutto diventa ancora più complicato perché spesso non si hanno gli script giusti per configurare correttamente gli ambienti. Inoltre, per garantire l'affidabilità e la disponibilità del servizio, si finisce spesso con il creare nuove macchine virtuali, o fisiche, per fare repliche e hosting di una sola applicazione, e questo implica che non solo ci saranno da gestire gli aggiornamenti applicativi ma bisognerà anche controllare gli aggiornamenti relativi al sistema operativo, installare patch di sicurezza, riconfigurare il web server allo stesso modo su tutte le macchine e così via. In ambienti in cui ci sono decine e decine di server dedicati, questo è un problema: non è sempre possibile avere script che ricreano la stessa configurazione su tutte le macchine, fare un deployment richiede tanto tempo in relazione al numero dei server e,

oltre a questo, ci si ritrova con tanto hardware che in realtà non fa nulla. Proprio per rimediare a questi problemi, si sviluppa il mondo dei container, capeggiato principalmente da aziende come Docker, il cui scopo è proprio quello di avere la virtualizzazione di una sola porzione della macchina virtuale, così da risparmiare sui costi, mantenere alta l'efficienza senza dover gestire il sistema operativo, creando un sistema isolato, il container, in grado di tenere al suo interno tutte le dipendenze, framework e configurazioni di cui il sistema stesso ha bisogno per funzionare.

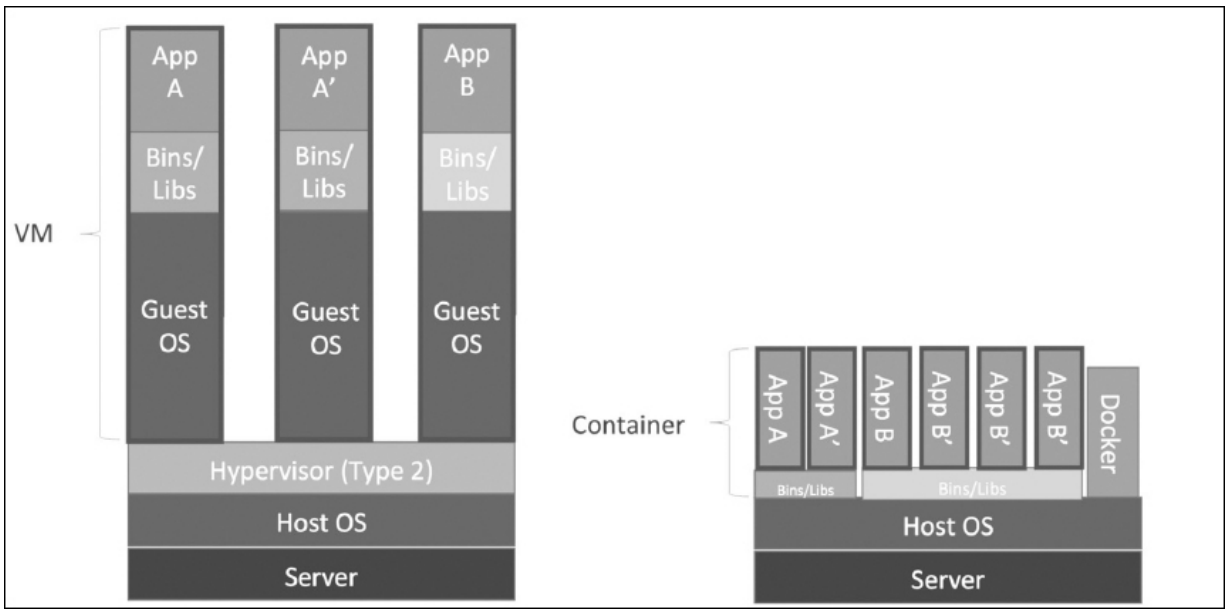


Figura 22.17 – Su una macchina virtuale c'è isolamento tra le applicazioni ma due istanze della stessa applicazione non possono condividere delle librerie in comune, perciò sono molto grandi, costose, inefficienti e complesse da gestire. I container, invece, vivono in isolamento e possono condividere le librerie attraverso un host (in questo caso il Docker Host), senza bisogno di un sistema operativo intermedio come nelle machine virtuali.

Come è dimostrato nella Figura 22.17, i container, non avendo più il sistema operativo delle VM da gestire e avendo la condivisione delle risorse e delle dipendenze, portano a una serie di vantaggi:

- Controllo granulare: l'approccio è orientato ai micro-servizi, anche se può funzionare con qualsiasi tipologia di applicazione.

- Testing facilitato: il testing (e il funzionamento) di un'applicazione in locale su container produrrà gli stessi risultati che in un qualsiasi altro ambiente, compresa la produzione.

- Deployment semplificato: l'applicazione è pacchettizzata con tutte le sue dipendenze in un solo componente, ovvero il container. In caso di rilascio di una nuova versione, i container possono girare in parallelo, consentendo anche la creazione di scenari di A/B testing.

- Avvio rapido: meno di un secondo, in genere, poiché c'è cache sulle immagini che creano i container.

Considerati tutti i vantaggi di questa nuova tipologia di sviluppo e deployment, Microsoft ha deciso di approcciarla facendo un'integrazione più semplice possibile direttamente all'interno di Visual Studio per i progetti .NET Core e ASP.NET Core. Nonostante cambi tutta la tecnologia, la semplicità nasce dal fatto che, tramite Visual Studio, il modello di sviluppo non cambia e si continuerà a lavorare come è stato sempre fatto, ma facendo il click con il tasto destro del mouse e selezionando Add -> Docker Support, come mostrato nella Figura 22.18, si avranno tutti i vantaggi illustrati nel paragrafo precedente.

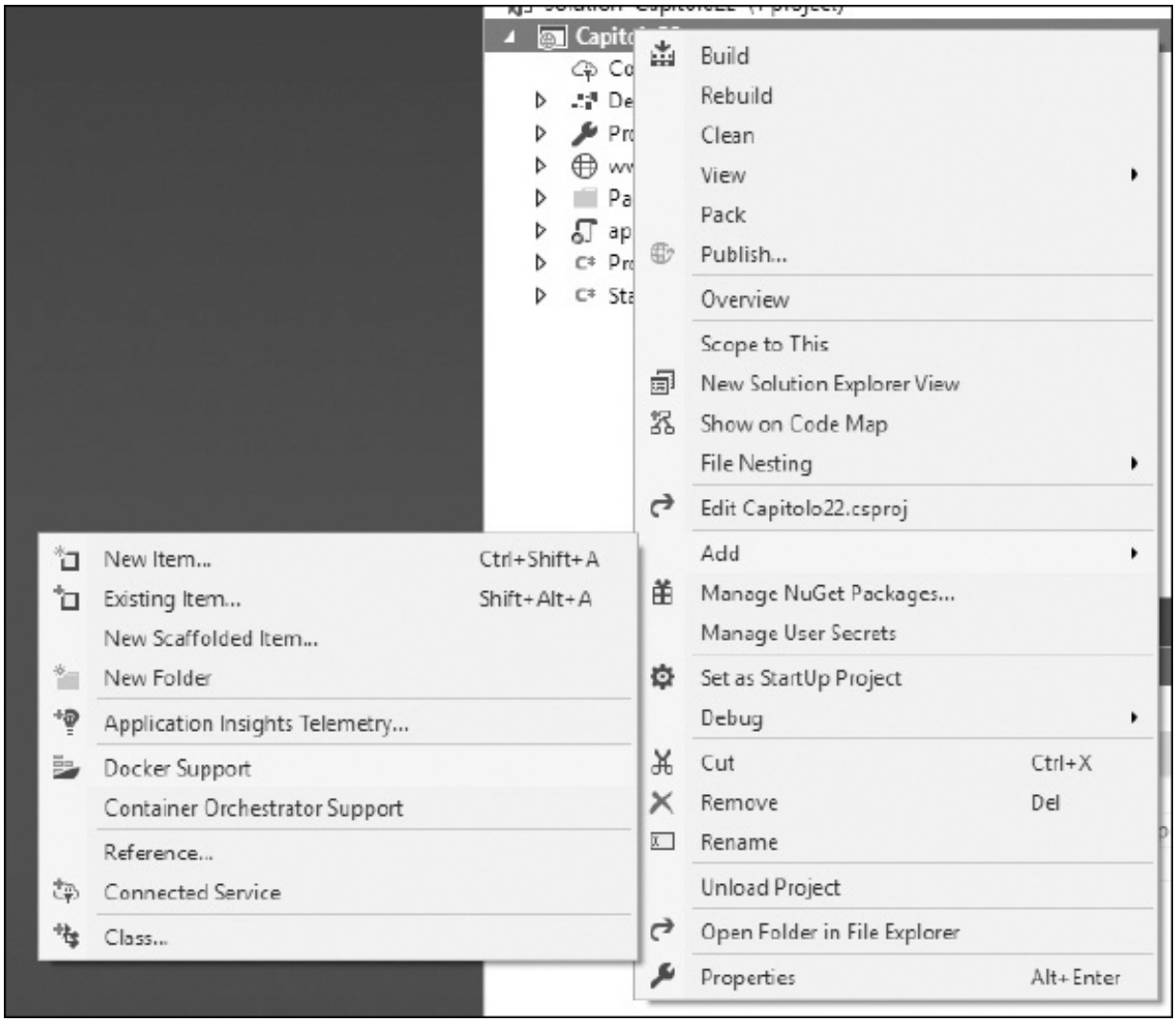


Figura 22.18 – L'aggiunta del supporto a Docker è fatta tramite il tasto destro sul progetto che si vuole containerizzare e selezionando l'opzione Add -> Docker Support.

Quello che avviene dopo questa operazione, mostrata nella Figura 22.18, è la creazione di un progetto, chiamato `docker-compose` a livello di solution, e di un file, chiamato `Dockerfile`, all'interno del progetto selezionato. Nella solution è stato creato un nuovo progetto, che contiene una serie di file, tra cui il `.dockerignore` che serve ai sistemi di source control per non fare il commit di alcuni tipi di file relativi a Docker, e il `docker-compose.yml`, un file con una sintassi molto simile al

JSON, in cui vengono descritti tutti i servizi che devono essere creati, come dimostra l'Esempio 22.17.

Esempio 22.17

```
version: ´3.4´
services:
  capitolo22:
    image: capitolo22
    build:
      context: .
      dockerfile: Capitolo22/Dockerfile
```

Nell'Esempio 22.17 si può notare come il servizio, in questo caso, sia solo uno perché il progetto sul quale abbiamo abilitato questa tecnologia è solamente uno e verrà creato a partire dal file `Dockerfile` contenuto nella sua soluzione. Quello che verrà creato non è il vero e proprio container ma solo una immagine che avrà il nome specificato nel tag image, ovvero "capitolo22": il container non sarà altro che l'esecuzione dell'immagine stessa; per questo, a livello di avvio e scalabilità, è molto più efficiente.

Il `Dockerfile`, ovvero il file di configurazione che specifica a Docker come deve essere costruita l'immagine, è stato creato di default sulla versione dell'SDK che abbiamo dichiarato a livello di progetto, in questo caso la versione 2.1 di ASP.NET Core. Pertanto il risultato ottenuto in automatico da Visual Studio sarà qualcosa di simile a quello mostrato nell'Esempio 22.18.

Esempio 22.18

```
FROM microsoft/dotnet:2.1-aspnetcore-runtime AS base
WORKDIR /app
EXPOSE 80

FROM microsoft/dotnet:2.1-sdk AS build
WORKDIR /src
COPY Capitolo22/Capitolo22.csproj Capitolo22/
RUN dotnet restore Capitolo22/Capitolo22.csproj
COPY . .
WORKDIR /src/Capitolo22
RUN dotnet build Capitolo22.csproj -c Release -o /app

FROM build AS publish
```

```
RUN dotnet publish Capitolo22.csproj -c Release -o /app

FROM base AS final
WORKDIR /app
COPY --from=publish /app .
ENTRYPOINT [”dotnet”, ”Capitolo22.dll”]
```

Nell’Esempio 22.18 viene mostrata una funzionalità particolare di Docker, chiamata multi-staged build e, infatti, si stanno susseguendo quattro fasi distinte dalla parola chiave `FROM`, che identifica un nuovo gruppo di operazioni:

- Fase 1: viene costruita una prima immagine, temporanea e nominata base, a partire da un’immagine già esistente e pre-configurata chiamata `dotnet`, che contiene il runtime di ASP.NET Core 2.1. Sebbene non sia obbligatorio, è molto utile per non partire dall’immagine vuota di Docker chiamata `SCRATCH`, in cui bisognerebbe lanciare comandi PowerShell o Bash in sequenza per installare tutto il runtime. Successivamente, viene creata una cartella chiamata app e quindi viene aperta la porta 80, che servirà a comunicare con l’esterno del container.

- Fase 2: viene costruita una seconda immagine, temporanea e nominata `build`, a partire da un’immagine già esistente chiamata sempre `dotnet`, che però questa volta contiene l’SDK di .NET Core 2.1, quindi vengono copiati i file di progetto e tutto il codice sorgente e, una volta impostata la working directory nella cartella corretta, viene lanciato il comando `dotnet build` con la configurazione di `Release`.

- Fase 3: viene costruita una terza immagine, sempre temporanea e nominata `publish`, a partire dall’immagine `build` creata nella fase 2, in cui avviene la pubblicazione tramite il comando `dotnet publish` in configurazione di `Release`, in cui viene anche specificato il percorso della cartella di output. Questo passaggio poteva anche essere inglobato con l’immagine generata in precedenza, ma per avere una maggiore separazione tra le varie operazioni è stata suddivisa in un’immagine a parte. Il comando `dotnet publish` è

possibile perché, dato che l'immagine di build è accessibile, lo saranno anche i sorgenti e i vari file di progetto in essa contenuti.

- Fase 4: viene costruita una quarta e ultima immagine a partire dall'immagine base costruita sul runtime nella Fase 1, in cui avviene la copia dei soli file contenuti nella cartella di pubblicazione dell'immagine costruita in Fase 3, così da non avere più i sorgenti ma solo le DLL compilate, quindi viene impostato l'entrypoint che sarà corrispondente al comando `dotnet Capitolo22.dll` che farà partire a tutti gli effetti l'applicazione ASP.NET Core.

Come possiamo notare, le immagini base dalla quale si è partiti sono due: il runtime e l'SDK. Poiché tutti i processi di build e pubblicazione vengono eseguiti direttamente all'interno del container, data la definizione del `Dockerfile`, c'è bisogno di un'immagine con tutto l'SDK per effettuare queste due operazioni, mentre quando si deve eseguire l'applicazione è sufficiente il runtime, che permette anche di risparmiare diverse centinaia di megabyte di spazio. L'operazione di build e pubblicazione all'interno del container stesso non è molto utile quando si lavora in un ambiente in cui c'è a disposizione Visual Studio ma è fondamentale in ambienti in cui non c'è installato .NET Core. È possibile vedere le immagini scaricate e la loro dimensione su disco lanciando il comando `docker images`, come dimostra la Figura 22.19.

L'immagine generata partendo dal `Dockerfile` dell'Esempio 22.19 sarà, come visibile nella Figura 22.19, di circa 250Mb, che, considerando un ambiente Linux, non è poco: ogni volta che si deve rilasciare un aggiornamento, viene potenzialmente spostata tutta l'immagine (anche se non è del tutto vero, dato che viene verificata la cache sui vari strati di cui è composta), quindi, più sarà grande, più tempo ci metterà a essere trasferita via network, più banda occuperà e più tempo ci metterà a partire in caso di scalabilità. Una delle novità introdotte a partire da .NET Core 2.1 riguarda l'introduzione di nuove immagini basate su Alpine, ovvero una variante di Debian, pensata per essere leggera. Infatti, l'immagine contenente il solo runtime arriva a pesare poco più di 80Mb, che rappresenta una differenza di circa 170Mb rispetto al passato. Per utilizzarla, sarà sufficiente modificare le istruzioni `FROM` dell'Esempio

22.18, impostando come immagini base `microsoft/dotnet:2.1-aspnetcore-runtime-alpine` per il runtime e `microsoft/dotnet:2.1-sdk-alpine` per l'sdk. Tutte le immagini che possono essere utilizzate per runtime e SDK sono visibili all'interno del Docker Hub di Microsoft, ovvero di un repository in cui sono salvate tutte le immagini pre-costruite e pronte da utilizzare, a partire da questo indirizzo: http://aspit.co/bo2.

```
C:\WINDOWS\system32\cmd.exe

C:\Users\Matteo>docker images
REPOSITORY                  TAG                         IMAGE ID        CREATED         SIZE
microsoft/dotnet            2.1-sdk-alpine              5fbf90294e33    4 days ago      1.46GB
microsoft/dotnet            2.1-aspnetcore-runtime-alpine 757184e1addb  4 days ago      163MB
microsoft/dotnet            2.1-sdk                     1e365a434b1b    4 days ago      1.73GB
microsoft/dotnet            2.1-aspnetcore-runtime      5b7745597818    4 days ago      255MB
microsoft/dotnet            core                        20305c46b164    3 months ago    253MB
microsoft/aspnetcore        latest                      f79840764591    3 months ago    302MB
microsoft/mssql-server-linux latest                     b0bb2e68f091    6 months ago    1.33GB
microsoft/aspnetcore-build  2.0                         61090cf9a0fb    8 months ago    1.85GB
microsoft/aspnetcore        2.0                         4f34e60fe340    8 months ago    280MB
redis                       4.0.2-alpine                5c7823413ff7    8 months ago    27.5MB
mysql                       latest                      141d24fea983    8 months ago    412MB
microsoft/aspnetcore        1.1                         7f07c27a5ec1    10 months ago   305MB
microsoft/dotnet            1.1.1-sdk                   82ba68a32f38    14 months ago   879MB
```

Figura 22.19 – Ogni immagine elencata dal comando `docker images` è composta da un nome, da un tag che ne specifica la versione, da un identificativo che la referenzia in modo univoco, da una data di creazione e da una dimensione.

Una volta dichiarato come deve essere costruita l'immagine che conterrà quindi l'applicazione e specificate le varie ottimizzazioni come l'uso di Alpine, verrà il momento di eseguirla. Osservando bene Visual Studio, dopo l'aggiunta del supporto a Docker, si noterà come è stato cambiato il progetto di startup, che non sarà più l'applicazione web stessa ma l'altro progetto `docker-compose` aggiunto da Visual Studio. Questo consente a Visual Studio di interagire con Docker e creare le immagini specificate nei `Dockerfile` a partire dalle specifiche dichiarate nel `docker-compose.yml` ma non solo: se in Docker è attiva la condivisione del disco, Visual Studio aprirà anche una porta per il debugger e, quindi, sarà possibile testare le applicazioni all'interno dei container stessi che di default sarebbero isolati. Qualora invece si volesse procedere

manualmente, sarà necessario lanciare solo due comandi, come viene mostrato nell'Esempio 22.19.

Esempio 22.19

```
docker build -t capitolo22 -f Capitolo22/Dockerfile .
docker run capitolo22
```

Come dimostrato nell'Esempio 22.19, infatti, il primo passaggio è quello di costruire l'immagine a partire dal `Dockerfile` creato in automatico da Visual Studio, con un nome specificato dal parametro `t`, quindi, una volta creata l'immagine, si potrà effettivamente avviare il container con il suo nome. A questo punto, l'applicazione ASP.NET Core verrà avviata con il comando definito dall'`ENTRYPOINT` del `Dockerfile` e ci sarà lo startup, come mostrato nella Figura 22.20.

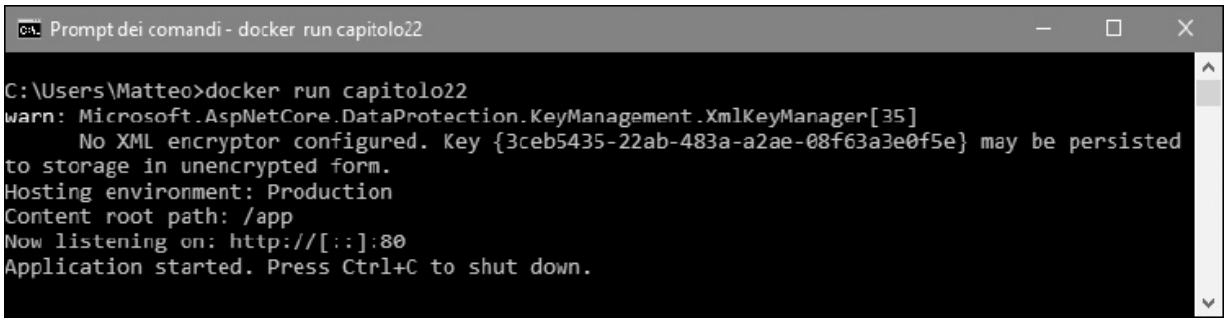


Figura 22.20 – L'avvio (manuale in questo caso) del container con l'applicazione ASP.NET Core farà partire Kestrel e aprirà la porta 80, esposta all'interno del container, per fare in modo che sia raggiungibile dall'esterno.

Una volta creata l'immagine e verificato il corretto funzionamento del container, si può procedere alla pubblicazione. Esistono diverse soluzioni che sono in grado di eseguire e gestire container, ma lo strumento più semplice è quello che abbiamo già visto in precedenza, ovvero gli App Service: la versione per Linux è infatti in grado di prendere l'immagine ed eseguirla, lanciando i comandi che abbiamo eseguito manualmente o tramite Visual Studio. Gli App Service sono comodi solo per un caso

molto specifico però, ovvero il caso in cui ci siano pochi servizi avviati come container, tutti definiti all'interno del file `docker-compose.yml` e tutti stateless. Nei casi in cui ci sia bisogno di avviare container che devono mantenere dati, piuttosto che container che necessitano di migliaia di istanze, gli App Service non rappresentano la soluzione migliore e, proprio per questo, in Azure sono disponibili altri sistemi come Azure Kubernetes Service (AKS), Service Fabric o Azure Container Instances (ACI) che permettono di mantenere cluster e ne garantiscono l'alta affidabilità e alta disponibilità. Maggiori informazioni su questi servizi sono disponibili su http://aspit.co/bo8.

Conclusioni

In questo capitolo abbiamo parlato di come stanno evolvendo le esigenze sia dei team di sviluppo sia dei team di operation e di come stanno sempre di più convergendo verso il mondo dei DevOps, in cui le competenze sono costituite da un insieme di software e infrastruttura, ma non solo: anche le tempistiche dei rilasci e la possibilità di avere più ambienti in cui verificare il funzionamento sono sempre di più punti critici nella crescita di un qualsiasi prodotto. Proprio per risolvere questi problemi, abbiamo visto e affrontato più nel dettaglio i concetti legati agli ambienti, integrati nativamente in ASP.NET Core. Sempre in tema di ambienti ma sul lato infrastrutturale, abbiamo visto come sfruttare il web server Kestrel in configurazione di reverse proxy con altri web server tra cui Apache, NGINX e il classico IIS e, in particolare, abbiamo discusso di come questo porti innumerevoli vantaggi in termini di prestazioni, sicurezza e semplicità di gestione. Poiché a oggi i costi dell'hardware sono spesso un problema e considerando anche che la scalabilità è spesso imprevedibile, si è affrontato il tema cloud con Microsoft Azure e il servizio gestito degli App Service, che permettono di fare hosting e offrire scalabilità automatica alle applicazioni ASP.NET Core, mentre in seguito si è discusso di Docker, che va ad accrescere in modo esponenziale questi concetti, portando anche grosse semplificazioni nella fase di deployment, in cui i tempi sono decisamente più stretti.

Con questo ultimo capitolo il nostro viaggio alla scoperta di ASP.NET Core è terminato. In questi ventidue capitoli, abbiamo analizzato le

nuove caratteristiche che ha portato, le peculiarità e i punti di forza di ASP.NET Core, analizzando in dettaglio la caratteristiche di ognuna delle sue funzionalità.

Vi abbiamo guidati attraverso un percorso che ha trattato le nozioni fondamentali, ponendo l'accento sulle novità e sulle caratteristiche più utili in un contesto in forte evoluzione, come quello rappresentato dal web. Speriamo di essere riusciti a trasmettervi almeno in parte la nostra passione e di avervi aiutato a costruire applicazioni web cross-platform, moderne e scalabili, basate su ASP.NET Core.

Buon lavoro e buon divertimento!

Informazioni sul Libro

Scritta per guidare gli sviluppatori alla scoperta di ASP.NET Core 2, il nuovo framework per il web cross platform e open source rilasciato da Microsoft, questa guida completa include tutte le ultime novità introdotte da ASP.NET Core e dalle tecnologie a corredo di applicazioni web, come Angular o l'accesso ai database.

Dalle basi di ASP.NET Core 2 ai concetti legati ad ASP.NET Core MVC, all'accesso ai dati, passando per identity e arrivando fino a JavaScript, Angular e tecnologie client-side, questo libro – con uno stile pratico e ricco di esempi – accompagna il lettore alla scoperta di tutte le caratteristiche che rendono ASP.NET Core uno dei toolkit più interessanti per sviluppare applicazioni web.

PUNTI DI FORZA

- Tutte le novità introdotte da ASP.NET Core
- I concetti base legati a ASP.NET Core MVC
- Gestione delle form con ASP.NET Core MVC
- Gestione dello stato
- Accesso ai dati con ADO.NET ed Entity Framework Core
- Sviluppo di servizi RESTful
- Gestione avanzata del runtime
- Globalizzazione e localizzazione
- Autenticazione e sicurezza delle applicazioni
- Sviluppo di applicazioni client-side e integrazione con ASP.NET Core